UCONN LIBRARY

University of Connecticut
OpenCommons@UConn

Master's Theses | University of Connecticut Graduate School

8-22-2011

AMIBE: an Imperative Programming Language with First Class Continuations

Yuting Wang
University of Connecticut - Storrs, yvting.wang@gmail.com

Recommended Citation
Wang, Yuting, "AMIBE: an Imperative Programming Language with First Class Continuations" (2011). *Master's Theses*. 142.
https://opencommons.uconn.edu/gs_theses/142

This work is brought to you for free and open access by the University of Connecticut Graduate School at OpenCommons@UConn. It has been accepted for inclusion in Master's Theses by an authorized administrator of OpenCommons@UConn. For more information, please contact opencommons@uconn.edu.

AMIBE: an Imperative Programming Language with First Class Continuations

Yuting Wang

B.E. Electric Engineering, Shanghai Jiao Tong University, China, 2006

M.E. Power System Automation, Shanghai Jiao Tong University, China, 2009

A Dissertation

Submitted in Partial Fullfilment of the

Requirements for the Degree of

Master of Science

at the

University of Connecticut

2011

APPROVAL PAGE

Master of Science Dissertation

AMIBE: an Imperative Programming Language with First Class Continuations

Presented by

Yuting Wang, B.E., M.E.

Major Advisor

Laurent Michel

Associate Advisor

Robert McCartney

Associate Advisor

Yufeng Wu

Associate Advisor

Zhijie Jerry Shi

University of Connecticut

2011

ACKNOWLEDGEMENTS

This thesis is the result of over one year of work with my advisor, Laurent Michel. I would like to thank him for the supports which make this research project possible. His enthusiasm and dedication gave me great confidence throughout the whole project. Advices from him (either for the research or for my personal life as a foreign student) greatly helped me in my graduate study. I also learned how to present my research clearly to others. This thesis would be much like a mindless hack rather than a well-organized story without his guidance.

I would also like to thank my committee members Robert McCartney, Zhijie Jerry Shi and Yufeng Wu for their comments and feedback. Steven Demurjian helped me a lot to get through the administrative process. Sanguthevar Rajasekaran and Alex Shvartsman gave me great help in my graduate study. I owe them all a debt of gratitude.

Many thanks are owed to Peter Luh and Guan Che on other projects that support my graduate school career. The research experiences I got by working with them are invaluable.

Last, I want to thank my family. My parents and grandparents supported my study abroad just like always. And many thanks to the rest of my family and friends who helped me in the past two years.

TABLE OF CONTENTS

AMIBE: an Imperative Programming Language with First Class Continuations

Yuting Wang, M.S.

University of Connecticut, 2011

A continuation represents the future of an execution. It is often used as an intermediate representation(IR) to compile functional programming languages, make control flow explicit and full beta-reduction(function inlining) possible. Continuations are also a language feature that gives user the ability to completely control the execution control flow(first class continuation). Efficient implementation of first class continuation is important for languages that need non-determinism and backtracking(e.g., COMET). We present a prototype imperative programming language with first class continuation – AMIBE. AMIBE uses the LLVM compiler infrastructure which is attractive for its optimizing tools and overall modern organization. However, LLVM does not support the implementation of continuation via a direct manipulation of the system stack. To move the execution state out of the system stack into a separate AMIBE stack, AMIBE adopts the Continuation Passing Style compilation technique(CPS). With CPS, states on the system stack are never reused since functions never return. Portable implementation for first class continuation becomes possible because the compiler only needs to save and restore the AMIBE stack which it fully controls. In CPS, function calls are tail calls. By exploiting the optimization for tail calls in LLVM, function calls are reduced to jumps, so that the system stack never grows on calls. AMIBE programs are first compiled into an AMIBE IR closely related to LLVM IR, then transformed into

CPS form. Finally the AMIBE IR in CPS is translated into LLVM IR. The performance of the optimizing compiler based on LLVM and CPS is compared against a naive just-in-time compiler based on GNU lightning and currently used by COMET.

Chapter 1

Introduction

1.1 Introduction to AMIBE

Continuations are often used as an intermediate representation to compile functional programming languages [21] [4] [15] [11] [10]. They are also useful as a language feature, which is essential to programming languages that need non-deterministic search and backtracking [8] [23] [20](for example, Constraint Programming Languages). We developed a prototype imperative programming language AMIBE that supports first class continuations. To exploit various optimizations conveniently (common expression elimination, constant folding, etc.) AMIBE uses Low Level Virtual Machine(LLVM), a compiler infrastructure that provides a collection of modular and reusable tool-chain technologies [19].

1.2 Background

Constraint Programming is a paradigm that attempts to reduce the gap between the high-level description of optimization problems and the computer algorithms implemented to solve them [25]. Constraint Programming consists of two part: Constraint and Search. Constraints are declarative constructs that captures the properties of the

problem. Search provides various non-deterministic search strategies to solve the problem (e.g., Depth First Search [9]). First class continuations are the foundation for the non-deterministic search [14]. They capture the control information (where the program is) and the execution state (what state the program is in) at some point. Later when search fails the program can backtrack with a first class continuation. COMET is such a constraint programming language [26]. It has its own virtual machine and intermediate representation, which means optimizations must be built from the ground up. AMIBE is an attempt to move COMET to a more flexible compiler infrastructure–LLVM. With LLVM a lot of well-known optimizations are possible such as:

- Peephole Optimization
- Common Sub-expressions Elimination
- Dead-Code Elimination
- Constant Propagation

1.3 Hello World in Constraint Programming: the N-Queen Problem

To realize the power of constraint programming language and the relation between non-deterministic search and first class continuations. We demonstrate how the classic N-QUEEN problem is programmed in COMET. In the N-QUEEN problem, a $N \times N$ chess board is given. A queen can only attack its enemy on the same row, column or diagonal. We need to find the placement of N queens on the chessboard such that none of them can attack each other. The N-QUEEN program in COMET is shown in Figure

1.1

```
1   import cotfd;
2   Solver<CP> cp();
3
4   int n = 8;
5   range S = 1..n;
6   var<CP>{int} q[i in S](cp, S);
7
8   Integer nSols(0);
9
10  solveall <cp> {
11    forall (i, j in S: i < j) {
12      cp.post(q[i] != q[j]);
13      cp.post(q[i]-q[j] != i-j);
14      cp.post(q[i]-q[j] != j-i);
15    }
16  } using {
17    label(q);
18    ++nSols;
19  }
20
21  cout << "Solution of " << n << "-Queens: " << nSols << endl;
```

Fig. 1.1: N-Queen program in Comet

At line 2, a CP solver *cp* is defined. Since every queen must be on a different row, an array of n CP variables is defined at line 6 to model the column positions of the queens on n different rows. $q[i]$ stores the columns in which the queen in row i could possibly be. Initially every element in q has domain $1..n$ which means they might be on any column from 1 to n.

The constraint solving program has the following structure:

```
1  solveall <cp> {
2    // post constraints
3  } using {
4    // non deterministic search
5  }
```

Lines 11 through 14 post the constraints applied on the CP variables. In the

N-QUEEN problem, for every pair of queens on rows i, j, the posted constraints says queen i cannot be on the same column as queen j and they cannot be on the diagonal of each other. The using block is lines 16-19 which define how non-deterministic search is performed. Constraints act on the domains of CP variables to remove inconsistent values. The non-deterministic search explores a search tree with a backtracking search algorithm (default algorithm is depth-first-search). At every node of the search tree filtering algorithms for the constraints are triggered to prune all values that do not participate in the solution. The filtering goes on until a fixed point is reached. At some node if it is not possible to get a solution (e.g., the domain of a variable is empty), the search must backtrack to a previous state and try another decision.

If a solution is found, the number of solutions $nSols$ increases and the search backtracks to find other solutions. If the domain of any element in q becomes empty, the search fails and backtracks to a previous state and tries another path. The non deterministic search statement $label(q)$ at line 17 is equivalent to:

```
1  forall (i in S)
2    tryall<cp>(v in S) cp.label(q[i], v);
```

The *tryall* statement above is equivalent to the function in Figure 1.2, which uses first class continuations to implement *tryall*.

A continuation statement

```
1  continuation c {
2     ....
3  }
4  // where the program should resume when c is called
```

binds the current continuation to c. Then the body of the continuation statement is evaluated and the normal execution continues. In COMET the continuation c keeps a

```
1   void tryall(Solver<CP> cp, var<CP>{int} q[i])
2   {
3     int v = q[i].getIMin(); // get the initial minimum of q[i]
4     int n = q[i].getIMax(); // get the initial maximum of q[i]
5     while (v <= n) {
6       int next = v+1;
7       bool firstContinuation = false;
8       continuation c {
9         if (next <= n) cp.push(c);
10        cp.label(q[i], v);
11        firstContinuation = true;
12      }
13      if (firstContinuation)
14        return;
15      else
16        v = next;
17    }
18  }
```

Fig. 1.2: *tryall* with continuation

snapshot of the COMET stack and remembers where the program should resume when it is called. When c is called the program returns to right after the continuation statement with the COMET stack restored.

In Figure 1.2, *tryall* goes through the domain of $q[i]$ $(1..n)$. At every iteration of the *while* statement, v is the current value and the next value to be checked is $v + 1$. At line 8, c is bound to a continuation which captures $next = v + 1$ and $firstContinuation = false$. The body of the continuation statement is evaluated, which pushes c onto the continuation stack if v does not exceed the upper bound of the domain(i.e., there are more values to try later). The *label* statement binds $q[i]$ to value v and $firstContinuation$ is set to *true*. Line 13 tests the $firstContinuation$ boolean to determine whether to return or try the next value. The first time it returns since $firstContinuation$ is *true*. When the saved continuation is called, $firstContinuation$

is restored to $false$ and the control flows to the next iteration of the loop.

The saved continuation might be called on a failure. A failure detected by the fix point algorithm triggers a call to the $fail$ method of the CP solver. Shown in Figure 1.3. If there are continuations on the stack, $fail$ pops out the most recent continuation can call it. Otherwise it calls the exit point continuation. When c is called the execution goes to line 13 in Figure 1.2 with stack value $next = v + 1$ and $firstContinuation = false$ restored. This time v is set to the next value $v + 1$ and another iteration of the $while$ statement starts which tries to bind $q[i]$ to $v + 1$.

```
1  void fail(Solver<CP> cp)
2  {
3    if (!cp.emptyStack()) {
4      Continuation c = cp.pop();
5      call(c);
6    }
7    else
8      exit();
9  }
```

Fig. 1.3: Fail in COMET

From the example we can see that a language with non-deterministic search needs first class continuations. With first class continuations it is easy to naturally express the backtrack strategy in the non-deterministic search.

1.4 Semantics of First Class Continuations in Comet

A first class continuation in COMET is a pair of instruction pointer and COMET stack state:

$< I, S >$ I=Instruction Pointer, S = COMET stack state (1.1)

```
1   continuation c {
2       ....
3   }
4   // where the program should resume when c is called
```

Fig. 1.4: Continuation Statement in COMET

When the continuation statement in Figure 1.4 is evaluated, the instruction pointer pointing to line 4 and the current COMET stack state are captured in the continuation c.

When c is called (as shown in Figure 1.5), the control flows to the instruction pointer saved in c (i.e., go to line 4 in Figure 1.4) with the COMET stack restored.

```
1   ...
2   call(c);
3   ...
```

Fig. 1.5: Continuation Call in COMET

Notice that a first class continuation in COMET only saves and restores the stack values. Heap values are not restored when a continuation is called. For instance, in Figure 1.2 $firstContinuation$ is a value on the stack which is saved in the continuation c and restored when c is called. In Figure 1.1 $nSols$ is a value on the heap which is not affected by first class continuation calls. The COMET stack is a user defined run-time structure which COMET fully controls. The state on the system stack is not captured in continuation. The motivation and technology to avoid saving and restoring the system

stack is discussed in the next section and in Chapter 3 in details.

1.5 Implementation of Fist Class Continuations in Comet

COMET uses GNU lightning, a non-optimizing compiler infrastructure. GNU lightning does not provide virtual machine instructions to save and restore the system stack. Instead, one must write assembly code to capture the state of the system stack which is not portable. COMET chooses to completely avoid leaving information on the system stack. It maintains its own stack – a user defined run-time structure to push and pop data by COMET compiler. The *call* and *return* instructions in GNU lightning are avoided since they leave states on the system stack. Instead function calls and returns are compiled directly in terms of jumps, pushing and popping on the COMET stack. In this way all the information about the execution state is on the COMET stack. The system stack is used only by library functions. To capture a continuation only the COMET stack needs to be saved. And restored when the continuation is called. Incremental algorithms for saving and restoring state are possible since the compiler fully controls the COMET stack. Since the function calls are actually jumps, the whole program becomes a large function with jumps across its body.

1.6 Implementation of Fist Class Continuations in AMIBE

AMIBE faces the same problem. It is desirable to completely avoid using the system stack since the LLVM compiler infrastructure does not provide virtual instructions to manipulate it. LLVM is a compiler infrastructure that provides various optimization

tools. The optimization tools are most effective on small functions. Therefore, it is desirable to adopt a strategy that does not result in a single large function, so that the LLVM optimizer can still be effective. In other words, real functions and function calls are needed in AMIBE. Still any information about the execution state should not be left on the system stack. One way to achieve this is to compile AMIBE programs with Continuation Passing Style(CPS). With CPS, functions never return which means states on the system stack are never reused. Hence they never need to be saved and restored by continuations. The execution state is kept on the AMIBE stack – a run-time structure the AMIBE compiler has full privilege to manipulate(just like the COMET stack). First class continuations are implemented by saving and restoring the AMIBE stack(Library calls still use the system stack).

Unlike most functional programming languages compiled with CPS, AMIBE does not use CPS as an intermediate representation(IR). Compiling function calls and returns with CPS is enough for the implementation of first class continuations. The AMIBE IR closely relates to the three address code virtual instructions in LLVM. A program is first translated into AMIBE IR with real function calls and returns. Then it goes through a CPS transformation which compiles the function calls into CPS form. Finally it is translated into LLVM instructions.

1.7 Structure of the Thesis

Chapter 2 defines the Small-step Operational Semantics for a simplified AMIBE language. The SOS semantics are used to guide the implementation of the AMIBE lan-

guage. Chapter 3 introduces the implementation of stack-based first class continuations in details. Chapter 4 develops the technology to transform AMIBE programs into CPS form. The partial CPS transformation for function calls and the implementation of first class continuations are discussed. Chapter 5 introduces the overall organization of the AMIBE compiler. Chapter 6 develops the front-end of the AMIBE compiler. The front-end takes AMIBE programs as input and outputs Abstract Syntax Trees. Chapter 7 defines the AMIBE IR virtual instructions and develops the IR code generation from AST. Chapter 8 gives the implementation of the CPS transformation in the AMIBE compiler. Chapter 9 discusses the translation of AMIBE IR into LLVM IR and optimizations in LLVM. Chapter 10 compares the performance of first class continuations in AMIBE with the implementation in COMET. Chapter 11 concludes the thesis and gives future research directions.

Chapter 2

AMIBE Grammar and Semantics

2.1 Introduction

AMIBE is designed to be a minimum imperative programming language with first class continuations which are extremely convenient for non-deterministic programming. The grammar of AMIBE is similar to conventional imperative programming languages such as C/C++/Java, with minimum extra grammar rules for continuations and closures. To give formal semantics to the AMIBE program, especially the semantics of first class continuations, we construct a baby AMIBE language in this chapter and build Small-step Operational Semantics(SOS) for it. The baby AMIBE is desirable for our discussion since it has simplified grammars while retaining major components of the AMIBE language.

In the following sections a baby language which has basic imperative control flow abilities such as function, sequencing, selection and iteration is built first. Then it is augmented with first class continuations and closures. The Small-step Operational Semantics for the baby language are given. The operational semantics guide us through the implementation issues of first class continuations in different compiler infrastructure,

such as GNU Lightning and LLVM.

2.2 Syntactic Conventions

The following syntactic conventions are used to describe the context free grammar of the baby AMIBE:

1. terminal symbols are in lower case and:
   (a) x denotes a name for an object(variable, function, etc.)
   (b) a denotes a function parameter
   (c) n denotes a integer constant
   (d) keywords are in **bold**
2. non-terminal symbols are in upper case and:
   (a) P denotes the whole program
   (b) F denotes a function
   (c) S denotes a statement
   (d) E denotes an expression
   (e) T denotes a type

2.3 Baby AMIBE: A Basic Imperative Language

The context free grammar of baby AMIBE is defined as follows:

A program consists of a list of functions. The *main* function is the entry point of the whole program. Like traditional imperative programming language, AMIBE drives

```
1   P : FL; x() //x is the main function
2
3   F : T x(T1 a1, T2 a2, ... , Tm am) {S}
4
5   FL : ;
6        | F;FL
7
8   S : ;
9       | S1 S2
10      | T x;
11      | E;
12      | if (E) S1 else S2
13      | while (E) S
14      | return E;
15
16  E : x
17      | n
18      | E1 + E2
19      | E1 < E2
20      | E1[E2]
21      | new T(E)
22      | E()
23      | E(ET)
24      | E1 = E2
25
26  ET : E
27      | E, ET
28
29  T : int
30      | bool
31      | T [ ]
```

Fig. 2.1: Context Free Grammar for Baby AMIBE

computation by evaluating statements that change the execution state. A function takes some number of inputs, evaluates its body(a list of statements) and returns a value. A simple factorial program is shown in Figure 2.2.

The program first defines the function $fact$ which computes factorial of the input n. In function $fact$ an "if" statement checks if n is equal to 0. If that is true a "return" statement 1. Otherwise it calls $fact(n-1)$ to get the factorial of $n-1$ and returns $n \times fact(n-1)$. The $main$ function definition follows. $main$ defines a local variable d

```
1   int fact(int n)
2   {
3     if (n==0)
4       return 1;
5     else
6       return fact(n-1)*n;
7   }
8
9   int main()
10  {
11    int d;
12    d = fact(9);
13    return 0;
14  }
```

Fig. 2.2: a Factorial Program in AMIBE

and calls $fact$ to compute the factorial of 9 and assigned the result to d. After that it returns 0.

From the example we can see the baby AMIBE is capable to do assignment and recursion, which makes it Turing Complete. Small-step Operational Semantics(SOS) are developed for major components of baby AMIBE: program, function, statement and expression in the following sections.

2.3.1 Semantic Domains

Small-step Operational Semantics(SOS) defines how individual steps of a computation takes places. It is one kind of Structural Operational Semantic which means it is syntax oriented and inductive. SOS consists of a set of reduction rules which not only specify what evaluations might be returned, but also a strategy to perform them [22].

To specify the SOS for baby AMIBE, the semantic domains must first be defined. The following semantic symbols are used in the definitions and examples:

1. n denotes an integer
2. l denotes a location
3. v denotes a value
4. lst denotes a list

The semantic domains and pre-defined functions on the domains are defined below:

Definition 2.3.1. *Boolean(Bool). A boolean value is either* $true$ *or* $false$.

$Bool = \{true, false\}$

Definition 2.3.2. *Integer(*Int*).*

$Int = \mathbb{Z}$

The functions on Int *are*

- $Add(n_1, n_2) : (Int \times Int) \to Int = n_1 +_{\mathbb{Z}} n_2$
- $Less(n_1, n_2) : (Int \times Int) \to Bool = n_1 <_{\mathbb{Z}} n_2$

Example 2.3.1. $-1, 3, 10000$ *are all integers.* $Add(3, 2) = 3 +_{\mathbb{Z}} 2 = 5$. $Less(1, -1) = 1 <_{\mathbb{Z}} -1 = false$.

Definition 2.3.3. *Location(Loc). A location is an address of the memory store.*

Definition 2.3.4. *Name(*$Name$*). A name denotes an object in an computation. It is represented by a string of characters:*

$Name = String$

Example 2.3.2. *n, $size1$, _cost are all names.*

Definition 2.3.5. *Environment(γ). An environment is a mapping from name to location*

$$\gamma = Name \mapsto Loc$$

Example 2.3.3. *$\gamma_1 = \{n \to l_1, size1 \to l_2\}$ is an environment that maps the name n to the location l_1 and the name $size1$ to the location l_2.*

Definition 2.3.6. *List(Lst). A ordered list of values of the same type. A list of type T can also be simply denoted as T^*. And a list of n elements is represented as $[v_1, v_2, ..., v_n]$*

The functions on $Lst = T^*$ are

- $Size_{lst}(lst) : Lst \to Int$ *// get the size of the list*
- $Get_{lst}(lst, n) : (Lst \times T) \to T$ *// get the n^{th} element in lst, where $0 \leq n < Size_{lst}(lst)$*

Example 2.3.4. *$lst_1 = [1, -2, 3, 5]$ is a list of integers. $lst_2 = [n, size1, \_cost]$ is a list of names. $Size_{lst}(lst_1) = 4$. $Get_{lst}(lst_2, 1) = size1$.*

Definition 2.3.7. *Array(Array). An array stores a vector of values of the same type. It is represented by a list of locations.*

$Array = Loc^*$

the functions on Array are

- $Array(locs) : Loc^* \to Array$ *// Constructor to create an array from a list of locations.*

- $Size_{ary}(Array(locs)) : Array \to Int = Size_{lst}(locs)$
- $Get_{ary}(Array(locs), i) : (Array \times Int) \to Loc = Get_{lst}(locs, i)$

Definition 2.3.8. *Function(Fun). A function takes a set of input values and returns an output. It is represented by its environment, a list of parameters and its body.*

$Fun = \gamma \times Name^* \times S$

the functions on Fun *are*

- *Constructor* $Fun(env, params, body) : (\gamma \times Name^* \times S) \to Fun$. *// Constructor*

Example 2.3.5. $f = Func(\{...\}, [n_1, n_2], return\ n_1 + n_2;)$ *is a function that takes input* n_1 *and* n_2*, computes the addition of them and returns the result.*

Definition 2.3.9. *Value(*Val*). A value is either a boolean, integer, location, array or function*

$Val = Bool + Int + Loc + Array + Fun$

Definition 2.3.10. *Store(*σ*). A store is a pair of mappings from location to value*

$\sigma = Loc \mapsto Val$

It consists of two parts: stack(α*) and heap(*β*). We use a pair* $< \alpha, \beta >$ *to represent* σ *when we deal with closures and continuations. The stack maps locations that are directly referred by names in the environment to values. The heap maps locations that are referred indirectly by names to values.*

Example 2.3.6. *In the environment* $\gamma_1 = \{n \to l_1, ary \to l_2\}$ *and the store* $\sigma_1 =< \alpha_1, \beta_1 >$ *where* $\alpha_1 = \{l_1 \to 3, l_2 \to Array([l_3, l_4])\}, \beta_2 = \{l_3 \to true, l_4 \to false\}$*, the stack* α_1 *stores the mappings* $l_1 \to 3$ *and* $l_2 \to Array([l_3, l_4])$ *where locations* l_1 *and* l_2

are directly referred by name n and ary in environment γ_1. Locations l_3, l_4 are locations in the array ary which can be get by first getting $Array([l_3, l_4])$ then using Get_{ary} method on it. They are referred indirectly by ary and their mappings are stored in the heap β_1.

Definition 2.3.11. *Tuple(Tup). A n-element tuple($< v_1, v_2, ..., v_n >$) is an ordered list of elements(usually of different types). It is defined recursively as follows:*

- $Tup_n = T_n \times Tup_{n-1}$
- $Tup_0 =<>$

Example 2.3.7. $< true, 1, Array([l_1, l_2]) >$ *is a 3-element tuple.*

2.3.2 Small-step Operational Semantics

SOS consists of a set of syntax-oriented reduction rules with the following format:

$$\frac{\text{antecedent}}{\text{consequent}} \tag{2.1}$$

where antecedent and consequent are relations with format:

$$\text{relation} = (\text{[context} \vdash\text{]proposition} \rightarrow \text{conclusion}) \tag{2.2}$$

For example, a reduction rule for the addition expression is shown below:

$$\frac{\gamma \vdash < \sigma, E_1 > \longrightarrow_e < \sigma', E_1' >}{\gamma \vdash < \sigma, E_1 + E_2 > \longrightarrow_e < \sigma', E_1' + E_2 >} \tag{2.3}$$

In the relation $\gamma \vdash < \sigma, E_1 > \longrightarrow_e < \sigma', E_1' >$ the environment γ is the context, $< \sigma, E_1 >$ is the proposition and $< \sigma', E_1' >$ is the conclusion. It can be interpreted as:

in the environment γ, expression E_1 can be evaluated to E_1' and the store changes from σ to σ'.

Rule 2.3 can be interpreted as: in environment γ with store σ, if an expression E matches the proposition of the consequent syntactically($E = E_1 + E_2$ in this case) and the antecedent can be verified inductively with SOS rules. E can be evaluated to $E_1' + E_2$ and the store will be changed to σ'.

2.3.3 Relations for Baby AMIBE

Relations for baby AMIBE are defined as follows:

Definition 2.3.12. *The domain of the relation for expressions is:*

$$\longrightarrow_e : \gamma \times (\sigma, E) \times (\sigma, E + Val) \tag{2.4}$$

Relations for expressions are:

$$\gamma \vdash < \sigma, E > \longrightarrow_e < \sigma', E' > \tag{2.5}$$

$$\gamma \vdash < \sigma, E > \longrightarrow_e < \sigma', v > \tag{2.6}$$

In 2.5, an expression E is evaluated in the environment γ to a simplified expression E' with the store σ changed to σ'.

In 2.6, an expression E is evaluated in the environment γ to a value v with the store σ changed to σ'.

Definition 2.3.13. *The domain of the relation for expression tuples is:*

$$\longrightarrow_{et} : \gamma \times (\sigma \times ET) \times (\sigma \times Tup) \tag{2.7}$$

The relation for the expression tuple is:

$$\gamma \vdash < \sigma, ET > \longrightarrow_{et} < \sigma', < v_0, v_1, ..., v_m >> \tag{2.8}$$

In 2.8, an expression tuple ET is evaluated in the environment γ to a tuple $< v_0, v_1, ..., v_m >$ with the store σ changed to σ'.

Definition 2.3.14. *The domain of the relation for getting the location of an expression E(if it is a L-value) is:*

$$\longrightarrow_l : \gamma \times (\sigma \times E) \times (\sigma \times Loc) \tag{2.9}$$

The relation is:

$$\gamma \vdash < \sigma, E > \longrightarrow_l < \sigma', l > \tag{2.10}$$

In 2.10, an expression E is evaluated in the environment γ to a location l with the store σ changed to σ'.

Definition 2.3.15. *The domain of the relation for statements is:*

$$\longrightarrow_s : (\gamma \times \sigma \times S) \times (\gamma \times \sigma \times (S + Val)) \tag{2.11}$$

The relations for statements are:

$$< \gamma, \sigma, S > \longrightarrow_s < \gamma', \sigma', S' > \tag{2.12}$$

$$< \gamma, \sigma, S > \longrightarrow_s < \gamma', \sigma', v > \tag{2.13}$$

In 2.12, a statement S is evaluated to an simplified statement S'. The environment γ changes to γ' and the store σ changes to σ'.

In 2.13, a statement S is evaluated to a value v which is the return value of the enclosing function(If S does not return a value, v is undefined($\perp$)). The environment γ changes to γ' and the store σ changes to σ'.

Definition 2.3.16. *The domain of the relation for functions is:*

$$\longrightarrow_f : (\gamma \times \sigma \times F) \times (\gamma \times \sigma) \tag{2.14}$$

The relation for functions is:

$$< \gamma, \sigma, F > \longrightarrow_f < \gamma', \sigma' > \tag{2.15}$$

In 2.15, the evaluation of the function F changes the environment γ and the store σ to γ' and σ' that contains the mapping of F.

Definition 2.3.17. *The domain of the relation for the function list is:*

$$\longrightarrow_{fl} : (\gamma \times \sigma \times FL) \times (\gamma \times \sigma) \tag{2.16}$$

The relation of the function list is:

$$< \gamma, \sigma, FL > \longrightarrow_{fl} < \gamma', \sigma' > \tag{2.17}$$

In 2.17, the evaluation of the function list FL changes the environment γ and the store σ to γ' and σ' that contains mappings of functions in FL.

Definition 2.3.18. *The domain of the relation for programs is:*

$$\longrightarrow_p : (\gamma \times \sigma \times P) \times (\gamma \times \sigma \times S) \tag{2.18}$$

The relation for programs is:

$$< \gamma, \sigma, P > \longrightarrow_p < \gamma', \sigma', main(); > \tag{2.19}$$

In 2.18, the evaluation of program P is simplified to evaluating the call to main in the environment γ' and the store σ'.

The following sections develops the SOS for baby AMIBE.

2.3.4 SOS for Expressions

From baby AMIBE's grammar, expressions include the following:

1. constant expression(n)
2. variable expression(x)
3. arithmetic expression($E_1 + E_2$)
4. relational expression($E_1 < E_2$)
5. array expression($E_1[E_2]$)
6. new expression(new $T(E)$)
7. function call expression($E(), E(ET)$)
8. assignment expression($E_1 = E_2$)

The SOS rule for the variable expression is Rule 2.20:

$$\frac{}{\gamma \vdash < \sigma, x > \longrightarrow_e < \sigma, \sigma(\gamma(x)) >} \tag{2.20}$$

The SOS rules for the addition expression($E : E_1 + E_2$) are Rules 2.21, 2.22 and 2.23. SOS for other arithmetic expressions are similar.

$$\frac{\gamma \vdash < \sigma, E_1 > \longrightarrow_e < \sigma', E_1' >}{\gamma \vdash < \sigma, E_1 + E_2 > \longrightarrow_e < \sigma', E_1' + E_2 >} \tag{2.21}$$

$$\frac{\gamma \vdash < \sigma, E_2 > \longrightarrow_e < \sigma', E_2' >}{\gamma \vdash < \sigma, E_1 + E_2 > \longrightarrow_e < \sigma', E_1 + E_2' >} \tag{2.22}$$

$$\frac{}{\gamma \vdash < \sigma, n_1 + n_2 > \longrightarrow_e < \sigma, Add(n_1 + n_2) >} \tag{2.23}$$

The SOS rules for the less-than expression($E : E_1 < E_2$) are Rules 2.24, 2.25 and 2.26. SOS for other relational expressions are similar.

$$\frac{\gamma \vdash < \sigma, E_1 > \longrightarrow_e < \sigma', E_1' >}{\gamma \vdash < \sigma, E_1 < E_2 > \longrightarrow_e < \sigma', E_1' < E_2 >} \tag{2.24}$$

$$\frac{\gamma \vdash < \sigma, E_2 > \longrightarrow_e < \sigma', E_2' >}{\gamma \vdash < \sigma, E_1 < E_2 > \longrightarrow_e < \sigma', E_1 < E_2' >} \tag{2.25}$$

$$\frac{}{\gamma \vdash < \sigma, n_1 < n_2 > \longrightarrow_e < \sigma, Less(n_1, n_2) >} \tag{2.26}$$

The SOS rules for the array expression($E : E_1[E_2]$) are Rules 2.27, 2.28, 2.29. Notice that the rules make sure the index expression is first evaluated to an integer, then the array expression is evaluated to an array, and finally the element in the store is retrieved. The strategy to evaluate the array expression is defined by these rules.

$$\frac{\gamma \vdash < \sigma, E_2 > \longrightarrow_e < \sigma', E_2' >}{\gamma \vdash < \sigma, E_1[E_2] > \longrightarrow_e < \sigma', E_1[E_2'] >} \tag{2.27}$$

$$\frac{\gamma \vdash < \sigma, E_1 > \longrightarrow_e < \sigma', E_1' >}{\gamma \vdash < \sigma, E_1[n] > \longrightarrow_e < \sigma', E_1'[n]) >} \tag{2.28}$$

$$\frac{}{\gamma \vdash < \sigma, v[n] > \longrightarrow_e < \sigma, \sigma(Get_{ary}(v, n)) >} \tag{2.29}$$

where v is an array

The new expression(E : new $T(E)$) allocates an array. Its SOS rules are Rules 2.30, 2.31.

$$\frac{\gamma \vdash < \sigma, E > \longrightarrow_e < \sigma', E' >}{\gamma \vdash < \sigma, \text{new } T(E) > \longrightarrow_e < \sigma', \text{new } T(E') >} \tag{2.30}$$

$$\frac{}{\gamma \vdash < \sigma, \text{new } T(n) > \longrightarrow_e < \sigma, Array(locs) >} \tag{2.31}$$

where $locs$ are a list of n locations allocated from the heap

The SOS rules for function call expressions($E : E()|E(ET)$) are Rules 2.32, 2.33, 2.36, 2.37. A function call evaluates its body in an environment augmented with parameters mapped to arguments. It returns a value as the result of the evaluation.

Rules 2.32, 2.33 evaluates function calls that do not take any arguments. The function expression E is first evaluated to a function $f = Fun(\gamma_f, [], S)$. Then the function body S is evaluated in the environment γ_f and the store σ. γ_f is the environment which f is defined in and σ is the current store. The return value v of evaluating S is the value of the function call.

$$\frac{\gamma \vdash < \sigma, E > \longrightarrow_e < \sigma', E' >}{\gamma \vdash < \sigma, E() > \longrightarrow_e < \sigma', E'() >} \tag{2.32}$$

$$\frac{< \gamma_f, \sigma, S > \longrightarrow_s < \gamma', \sigma', v >}{\gamma \vdash < \sigma, f() > \longrightarrow_e < \sigma', v >} \tag{2.33}$$

where $f = Fun(\gamma_f, [], S)$

For function calls that take arguments, the arguments are evaluated first by Rules 2.34, 2.35. The result is a tuple containing the evaluated values.

$$\frac{\gamma \vdash < \sigma, E > \longrightarrow_e < \sigma', v >}{\gamma \vdash < \sigma, E > \longrightarrow_{et} < \sigma', < v >>} \tag{2.34}$$

$$\frac{\gamma \vdash < \sigma, E > \longrightarrow_e < \sigma', v_0 >, \gamma \vdash < \sigma', ET > \longrightarrow_{et} < \sigma'', < v_1, ..., v_m >>}{\gamma \vdash < \sigma, E, ET > \longrightarrow_{et} < \sigma'', < v_0, v_1, ..., v_m >>} \tag{2.35}$$

Rules 2.36, 2.37 evaluates function calls that take arguments. Arguments are evaluated into a tuple $< v_1, ..., v_m >$ first. Then the function expression E is evaluated to a function $f = Fun(\gamma_f, [a_1, ..., a_m], S)$. In rule 2.37 the store σ is represented by a pair of stack α and heap β. γ_f is extended to γ_1 that contains mappings from parameter names $a_1, .., a_m$ to the locations $l_1, .., l_m$. α is extended to α_1 that contains mappings from $l_1, ..., l_m$ to arguments values $v_1, .., v_m$. The function body S is evaluated with γ_1, α_1 and β. After that the store changes to σ'. The return value v of evaluating S is the value of the function call.

$$\frac{\gamma \vdash < \sigma, E > \longrightarrow_e < \sigma', E' >}{\gamma \vdash < \sigma, E(< v_1, ..., v_m >) > \longrightarrow_e < \sigma', E'(< v_1, ..., v_m >) >} \tag{2.36}$$

$$\frac{< \gamma_1, < \alpha_1, \beta >, S > \longrightarrow_s < \gamma', \sigma', v >}{\gamma \vdash << \alpha, \beta >, f(< v_1, ..., v_m >) > \longrightarrow_e < \sigma', v >}$$

where $f = Fun(\gamma_f, [a_1, ..., a_m], S)$ and

$\gamma_1 = \gamma_f[a_1 \mapsto l_1][a_2 \mapsto l_2]...[a_m \mapsto l_m]$ and (2.37)

$\alpha_1 = \alpha[l_1 \mapsto v_1][l_2 \mapsto v_2]...[l_m \mapsto v_m]$

where $l_i (1 \leq i \leq m)$ are allocated from the stack α

An assignment to an expression is to associate a location to a new value. In the baby AMIBE only variable name and array elements can be assigned to. The SOS rules for getting their locations are:

$$\frac{}{\gamma \vdash < \sigma, x > \longrightarrow_l < \sigma, \gamma(x) >} \tag{2.38}$$

$$\frac{\gamma \vdash < \sigma, E_2 > \longrightarrow_e < \sigma', n >, \gamma \vdash < \sigma', E_1 > \longrightarrow_e < \sigma'', v >}{\gamma \vdash < \sigma, E_1[E_2] > \longrightarrow_l < \sigma'', Get_{ary}(v, n) >} \tag{2.39}$$

where v is an array

The SOS rules for the assignment expression($E_1 = E_2$) are Rules 2.40, 2.41. The right hand side is evaluated first to value v. The left hand side is evaluated to an address l and the assignment produces an store with l mapped to v.

$$\frac{\gamma \vdash < \sigma, E_2 > \longrightarrow_e < \sigma', E_2' >}{\gamma \vdash < \sigma, E_1 = E_2 > \longrightarrow_e < \sigma', E_1 = E_2' >} \tag{2.40}$$

$$\frac{\gamma \vdash < \sigma, E_1 > \longrightarrow_l < \sigma', l >}{\gamma \vdash < \sigma, E_1 = v > \longrightarrow_e < \sigma'[l \mapsto v], v >} \tag{2.41}$$

2.3.5 SOS for Statements

Statements include the following:

1. skip statement(;)
2. sequential statements(S_1 S_2)
3. declaration statement(T x;)
4. expression statement(E;)
5. selection statement(if (E) S_1 else S_2)
6. iteration statement(while (E) S)
7. return statement(return E;)

The SOS rule for the skip statement(;) is Rule 2.42

$$\frac{}{< \gamma, \sigma, ; > \longrightarrow_s < \gamma, \sigma, \perp >} \tag{2.42}$$

The SOS rules for the sequential statement($S_1\ S_2$) are Rules 2.43, 2.44. The statement is evaluated sequentially. If a statement returns a value v then the following statements will not be evaluated.

$$\frac{< \gamma, \sigma, S_1 > \longrightarrow_s < \gamma', \sigma', S_1' >}{< \gamma, \sigma, S_1\ S_2 > \longrightarrow_s < \gamma', \sigma', S_1'\ S_2 >} \tag{2.43}$$

$$\frac{< \gamma, \sigma, S_1 > \longrightarrow_s < \gamma', \sigma', v >}{< \gamma, \sigma, S_1\ S_2 > \longrightarrow_s < \gamma', \sigma', (v == \perp ? S_2 : v) >} \tag{2.44}$$

The SOS rule for the declaration statement($T\ x;$) is Rule 2.45. The environment maps variable x to a new address l and the store maps l to an undefined value.

$$\frac{}{< \gamma, \sigma, \mathrm{T}\ x; > \longrightarrow_s < \gamma[x \mapsto l], \sigma[l \mapsto \perp], \perp >} \tag{2.45}$$

where l is allocated from the stack

The SOS rules for the expression statement($E;$) are Rules 2.46 2.47. The expression SOS rules are used for the evaluation.

$$\frac{\gamma \vdash < \sigma, E > \longrightarrow_e < \sigma', E' >}{< \gamma, \sigma, E; > \longrightarrow_s < \gamma, \sigma', E'; >} \tag{2.46}$$

$$\frac{}{< \gamma, \sigma, v; > \longrightarrow_s < \gamma, \sigma, \perp >} \tag{2.47}$$

The SOS rules for the selection statement(if $(E)\ S_1$ else S_2) are Rules 2.48, 2.49, 2.50. The conditional expression is evaluated first. If it is true S_1 is evaluated. Otherwise S_2 is evaluated.

$$\frac{\gamma \vdash < \sigma, E > \longrightarrow_e < \sigma', E' >}{< \gamma, \sigma, \text{ if } (E) \ S_1 \text{ else } S_2 > \longrightarrow_s < \gamma, \sigma, \text{if } (E') \ S_1 \text{ else } S_2 >} \tag{2.48}$$

$$\frac{}{< \gamma, \sigma, \text{if } (true) \ S_1 \text{ else } S_2 > \longrightarrow_s < \gamma, \sigma, S_1 >} \tag{2.49}$$

$$\frac{}{< \gamma, \sigma, \text{if } (false) \ S_1 \text{ else } S_2 > \longrightarrow_s < \gamma, \sigma, S_2 >} \tag{2.50}$$

The SOS rule for the iteration statement(while (E) S) is Rule 2.51. It is transformed into a selection statement. If the conditional expression E is true then S is looped. Otherwise the loop ends.

$$\frac{< \gamma, \sigma, \text{if } (E) \ \{S; \text{ while } (E) \ S\} \text{ else } ; > \longrightarrow_s < \gamma', \sigma', S' >}{< \gamma, \sigma, \text{while } (E) \ S > \longrightarrow_s < \gamma', \sigma', S' >} \tag{2.51}$$

The SOS rules for the return statement are Rules 2.52, 2.53. The expression E is evaluated and its value is the return value to the enclosing function.

$$\frac{\gamma \vdash < \sigma, E > \longrightarrow_e < \sigma', E' >}{< \gamma, \sigma, \text{return } E; > \longrightarrow_s < \gamma, \sigma', \text{return } E'; >} \tag{2.52}$$

$$\frac{}{< \gamma, \sigma, \text{return } v; > \longrightarrow_s < \gamma, \sigma, v >} \tag{2.53}$$

2.3.6 SOS for Functions

The evaluation of a function creates a new function object. The SOS rule for function definition($T \ x(T_1 \ a_1, \ T_2 \ a_2, \ \ldots \ , \ T_n \ a_n) \ \{S\}$) is Rule 2.54. A new location l is allocated from the store and x maps to this location which maps to a function object in the store. Notice the environment of the function is undefined right now. It will be set to the environment when all the function names are visible, so that mutual recursive functions are possible.

$$\frac{}{< \gamma, \sigma, T\ x(T_1\ a_1,\ T_2\ a_2,\ ...\ ,\ T_m\ a_m)\ \{S\}\ > \longrightarrow_f < \gamma[x \mapsto l], \sigma[l \mapsto \text{Fun}(\perp, [a_1, ..., a_m], S)] >}$$

where l is allocated from the heap

(2.54)

2.3.7 SOS for Programs

A program consists of a list of function definitions and a call to the main function. The SOS rules for the function list$(FL :; |F; FL)$ are Rules 2.55, 2.56. The function list are evaluated sequentially. When the end of the list is reached. The environments of all functions are replaced by the current environment in which all the function names are visible (as shown in Rule 2.55). So that mutual recursive calls are possible.

$$\frac{}{< \gamma, \sigma, ; > \longrightarrow_{fl} < \gamma, \sigma' >} \tag{2.55}$$

where $\sigma' = \{[l \mapsto \text{Fun}(\gamma, [a_1, ..., a_m], S)] \mid [l \mapsto \text{Fun}(\perp, [a_1, ..., a_m], S)] \in \sigma\}$

$$\frac{< \gamma, \sigma, F > \longrightarrow_f < \gamma_1, \sigma_1 >, < \gamma_1, \sigma_1, FL > \longrightarrow_f < \gamma', \sigma' >}{< \gamma, \sigma, F; FL > \longrightarrow_{fl} < \gamma', \sigma' >} \tag{2.56}$$

The SOS rule for the program$(P : FL; x())$ is Rule 2.57. A list of functions are defined which creates the global environment γ_g and store σ_g. Then the call to the main function x is evaluated.

$$\frac{< \gamma, \sigma, FL > \longrightarrow_{fl} < \gamma_g, \sigma_g >}{< \gamma, \sigma, FL; x(); > \longrightarrow_p < \gamma_g, \sigma_g, x(); >} \tag{2.57}$$

where $\sigma_g(\gamma_g(x)) = \text{Fun}(\gamma_g, [], S)$

2.4 Extension of Closures and Continuations to the Baby AMIBE Language

The context free grammar for continuations and closures is added to the baby AMIBE language as follows:

```
1   S: ....
2   | closure x {S}
3   | continuation x {S} PC
4   | call(x);
5
6   T: ...
7   | Closure
8   | Continuation
```

Fig. 2.3: Context Free Grammar for Continuations and Closures in Baby AMIBE

1. The closure statement(closure x $\{S\}$) defines a closure x with body S. A closure captures the current environment and the stack. When it is called the statement S is evaluated in the captured environment and stack.

2. The continuation statement (continuation x $\{S\}$ PC) captures the current environment and the stack in the continuation x and execute the statement S. When x is called, the execution go to right after the continuation statement (at PC) with the captured environment and stack restored. The PC is an instruction pointer which tells where the execution should go when the continuation is called.

3. The call statement(call (x);) is overloaded for closure and continuation calls. If x is a closure, the closure is called. If x is a continuation, the continuation is called.

An example for first class continuations appears in Figure 2.4:

```
1   int fact(int n, Continuation[] x)
2   {
3     if (n == 0) {
4       continuation c {
5         x[0] = c;
6       }
7       return 1;
8     }
9     else
10      return n*fact(n-1, x);
11  }
12
13  int main()
14  {
15    Continuation[] x = new Continuation[](1);
16    int d = fact(5, x);
17    call(x[0]);
18  }
```

Fig. 2.4: A Simple Continuation Example in AMIBE

At line 15, an array x containing a single continuation is defined. As mentioned before array elements are on the heap. Thus the value in x will not be affected by continuation calls. Line 16 calls $fact(5, x)$ to compute the factorial of 5. $fact$ computes factorial recursively, except that when $n = 0$ a continuation c is created and stored in the array x. Continuation c saves the environment, the stack and the instruction pointer after the continuation statement(i.e., instruction pointing to line 7). After $fact(5, x)$ returns 120, line 17 calls the continuation $x[0](x[0] = c)$.The control flows back to line 7 with the environment and the stack in c restored. The result of recursively computing the factorial of 5 is returned once again(i.e., $fact(5, x)$ at line 16 returns 120). When line 17 is reached $x[0]$ is called once again. The same process goes over and over again. Thus this example computes the factorial of 5 infinitely.

2.4.1 Semantic Domains

The semantic domains for closures and continuations are defined as follows:

Definition 2.4.1. *Closure(Clos). A closure captures the current environment and the stack*

The functions on Clos are:

- $Clos(env, stack, body) : (\gamma \times \alpha \times S) \rightarrow Clos$ *//Constructor*

Definition 2.4.2. *Continuation(Cont). A continuation captures an instruction pointer PC, the environment and the stack. When called it goes back to the instruction pointed to by PC with the environment and stack restored.*

The functions on Cont are:

- $Cont(env, stack, pc) : (\gamma \times \alpha \times PC) \rightarrow Cont$ *// Constructor*

The Val domain in Definition 2.3.9 is redefined to include $Clos$ and $Cont$ below:

Definition 2.4.3. *Value(Val).*

$Val = Bool + Int + Loc + Array + Fun + Clos + Cont$

2.4.2 SOS for Closures

The SOS rules for the closure statement and call are Rules 2.58, 2.59. The store σ is represented with a pair of stack α and heap β. Only α is captured in the closure when it is defined in Rule 2.58. The call to the closure evaluates S with the stack and environment restored in Rule 2.59. After the evaluation the change on the heap(from β to β') is kept.

$$\frac{}{< \gamma, < \alpha, \beta >, \text{closure } x\ \{S\}; > \longrightarrow_s < \gamma[x \mapsto l], < \alpha[l \mapsto \text{Clos}(\gamma, \alpha, S)], \beta >, \perp >}$$

where l is allocated on the stack α

(2.58)

$$\frac{< \gamma_c, < \alpha_c, \beta >, S > \longrightarrow_s < \gamma_c', < \alpha', \beta' >, v >}{< \gamma, < \alpha, \beta >, \text{call } (x); > \longrightarrow_s < \gamma, < \alpha, \beta' >, \perp >} \quad \text{where } \sigma(\gamma(x)) = \text{Clos}(\gamma_c, \alpha_c, S)\ (\sigma =< \alpha, \beta >) \tag{2.59}$$

2.4.3 SOS for Continuations

The SOS rules for continuation statement and call are Rules 2.60, 2.61. In Rule 2.60 a first class continuation x is created at the continuation statement. The body S is evaluated, which might save x in the heap for later use. Notice that the antecedent is semantically the same to a call to the anonymous function (*void* (Continuation x) $\{S\}$).

Rule 2.61 defines a pseudo-SOS for continuation calls. If x represents a continuation $\text{Cont}(\gamma_c, \alpha_c, PC)$, the call statement "call (x)" is reduced to a state $< \gamma_c, < \alpha_c, \beta >, PC >$, which unfortunately does not match exactly the reduction rules for statements. PC is an instruction pointer pointing to the instruction following the continuation statement. The reduction of $< \gamma_c, < \alpha_c, \beta >, PC >$ evaluates the future instructions starting from PC with in environment γ_c and store$< \alpha_c, \beta >$. Thus the reduction of $< \gamma_c, < \alpha_c, \beta >, PC >$ is not a local reduction but a reduction from a state where reduction steps involving PC have not yet been finished. If we think SOS as an evaluator, the reduction of $< \gamma_c, < \alpha_c, \beta >, PC >$ simply means the evaluation jumps to the instruction pointed by PC with the environment restored to γ_c and the stack

restored to α_c. (We don't give a strict syntax-oriented SOS because it requires defining all rules with CPS which deviates from the conventional imperative semantics). Like the closure a continuation does not save the heap.

$$\frac{< \gamma[x \mapsto l], < \alpha[l \mapsto \text{Cont}(\gamma, \alpha, PC)], \beta >, S > \longrightarrow_s < \gamma', \sigma', v >}{< \gamma, < \alpha, \beta >, \text{continuation } x \ \{S\};\ PC > \longrightarrow_s < \gamma, \sigma', \perp >} \quad \text{where } l \text{ is allocated on the stack} \tag{2.60}$$

$$\frac{}{< \gamma, < \alpha, \beta >, \text{call } (x); > \longrightarrow_c < \gamma_c, < \alpha_c, \beta >, PC >} \quad \text{where } \sigma(\gamma(x)) = \text{Cont}(\gamma_c, \alpha_c, PC)\ (\sigma = < \alpha, \beta >) \tag{2.61}$$

2.4.4 Library Functions for Non-Deterministic Search Models

With closures and first class continuations the baby AMIBE is already capable of solving a variety of non-deterministic search problems. These problems are often hard to model and solve in languages without first class continuations. Some library functions for non-deterministic search are defined in this section. They are fully reusable from one model to another.

The library functions to load and store values from heap are defined in Figure 2.5. Remember values on the heap are not restored in a continuation call. If we want side effects that are permanent then it should happen on the heap. Since AMIBE does not allow global variables, a one element array is used to represent and store a single integer on the heap(from Rule 2.31 we know elements in the array are stored on the heap).

A continuation stack is used to keep track of the possible branches when perform-

```
1   int [] createHeapValue()
2   {
3     return new int[](1);
4   }
5
6   void setHeapValue(int[] var, int v)
7   {
8     var[0] = v;
9   }
10
11  int getHeapValue(int[] var)
12  {
13    return var[0];
14  }
15
16  void incHeapValue(int[] var)
17  {
18    var[0] = var[0]+1;
19  }
20
21  void decHeapValue(int[] var)
22  {
23    var[0] = var[0]-1;
24  }
```

Fig. 2.5: Functions to Load, Store and Compute Heap Values

ing a depth first search. It is a pair of a continuation array and its size. The *pushStack* and *popStack* functions are defined in Figure 2.6.

```
1   void pushStack(Continuation[] contStack, int[] size, Continuation cont)
2   {
3     contStack[getHeapValue(size)] = cont;
4     incHeapValue(size);
5   }
6
7   Continuation popStack(Continuation[] contStack, int[] size)
8   {
9     Continuation cont = contStack[getHeapValue(size)-1];
10    decHeapValue(size);
11    return cont;
12  }
```

Fig. 2.6: Functions to Push and Pop the Continuation Stack

The function *label* is used to mark a value n with domain $[l, u]$(shown in Figure 2.7). The first time it is called it returns the lower bound $n = l$. Each time the program backtrack to *label* it returns $n = n+1$, until n is larger than u. Naturally, n is allocated on the heap. Same for the continuation *cont*. If $n < u$ (i.e. n has not exceed the upper bound) the continuation *cont* is pushed onto the stack for its future use when backtracking.

```
1   int label(int l, int u, Continuation[] contStack, int[] stackSize)
2   {
3     // allocate it on heap so that value is kept after continuation call
4     int[] n = createHeapValue();
5
6     bool first = false;
7     Continuation[] cont = new Continuation[](1);
8     continuation x {
9       // store the continuation
10      cont[0] = x;
11       first = true;
12    }
13
14    if ( first ) setHeapValue(n, l); else incHeapValue(n);
15
16    int v = getHeapValue(n);
17    if (v < u) {
18      // push continuation because there are more values to label later
19      pushStack(contStack, stackSize, cont[0]);
20    }
21    return v;
22  }
23
24
25  void backtrack(Continuation[] contStack, int[] stackSize)
26  {
27    if (getHeapValue(stackSize) == 0) return;
28    else {
29      Continuation cont = popStack(contStack, stackSize);
30      call(cont);
31    }
32  }
```

Fig. 2.7: Label and Backtrack Functions in AMIBE

The function *backtrack* backtracks with the top continuation on the continuation stack. It pops out the top continuation from the stack and calls it. Combining with *label* and *backtrack* we would be able to search integer values over finite domains.

2.4.5 N-Queen problem in AMIBE

The N-QUEEN program is shown in Figure 2.8.

On a $n \times n$ chessboard n queens should be on different rows to avoid attacking each other. An array q of n integers is used to represent which column the queen on each row is in. For instance if $q[2] = 4$ the queen on row 2 is in column 4. The *attack* function checks if queen on row i and j can attack each other. The *queen* function takes an array of queens and a continuation stack. The outer for statement goes through the n rows and places a queen on each row. The call to the *label* function tries to get a column on which the i^{th} queen should be placed. Then it checks any queens that has already been placed are not attacking the i^{th} queen. If not, it backtracks to try a new placement. It is a depth first search on the columns of every queen. In the *main* function a value *nSols* representing the number of solutions is created on the heap. Then *main* calls the *queen* function. If it successfully places all of the queens(*queen* returns) then the number of solutions is increased by 1. It then backtracks to find more solutions. When all of the solutions have been found it hits the exit point continuation and prints out the number of solutions. In this example $n = 8$ and the program prints out 92.

From the example we can see it is very easy to write programs involving non-deterministic search in AMIBE because it supports first class continuations. To implement first class continuations AMIBE must be able to save and restore stacks. The

next chapter discusses how the stack saving and restoring are implemented in COMET with GNU Lightning. Then we move onto the implementation in AMIBE with LLVM. The discussion focuses on the implementation of first class continuations. The implementation of closures is similar and much easier as it only needs to save and restore local information on the stack.

```
1   // queen i attacks queen j
2   bool attack(int[] q, int i, int j)
3   {
4     return q[i]==q[j] || q[i]-q[j]==i-j || q[i]-q[j]==j-i;
5   }
6
7   void queen(int[] q, Continuation[] contStack, int[] stackSize)
8   {
9     int n = q.length;
10    for (int i = 0; i < n; i=i+1) {
11      q[i] = label(0, n-1, contStack, stackSize);
12      for (int j=0; j < i; j=j+1) {
13        if (attack(q, i, j))
14          backtrack(contStack, stackSize);
15      }
16    }
17  }
18
19  int main()
20  {
21    int MAX_STACK_SIZE = 50;
22    Continuation[] contStack = new Continuation[](MAX_STACK_SIZE);
23    int[] stackSize = createHeapValue();
24
25    int n = 8;
26    int[] nSols = createHeapValue();
27
28    bool exit = true;
29    continuation exitPoint {
30      pushStack(contStack, stackSize, exitPoint);
31      exit = false;
32    }
33
34    if (!exit) {
35      int[] q = new int[](n);
36      queen(q, contStack, stackSize);
37      incHeapValue(nSols);
38      backtrack(contStack, stackSize);
39    }
40    else {
41      printi(getHeapValue(nSols));
42      return 0;
43    }
44  }
```

Fig. 2.8: a N-Queen Program in AMIBE

Chapter 3

Stack Based Continuations

3.1 Stack Based Continuations in Imperative Programming Languages

To implement first class continuations a language must be able to save and restore stacks. In most imperative programming languages the stack stores values of local variables. Values that are referred indirectly by local variables are on the heap, including objects, array elements, etc. The stack is a first in first out(FIFO) structure. A frame is allocated on the stack when a function is called and the stack grows. The frame is deallocated when the function returns and the stack shrinks.

On compiler infrastructures that allow explicit manipulation of the system stack it is straight forward to implement first class continuations. When a first class continuation is created by a continuation statement, it takes a snapshot of the stack and remembers the address of the instruction following the continuation statement. When the continuation is called, the program restores the stack snapshot. Then it jumps to the instruction following the continuation statement.

Figure 3.1 is an example of the factorial program with first class continuations. The *main* function calls $fact$ to compute the factorial of 5. $fact$ computes factorial

recursively. When n reaches 0, the continuation statement in $fact$ captures the current stack in c and c is assigned to a global variable x. The program then continues the normal control flow and returns 1. The factorial of 5 is printed. When the $call(x)$ statement at line 18 is reached the continuation c is called and program goes to line 9 with the stack captured in c restored. The program repeats the computation of the factorial of 5 again and prints it out. When it hits $call(x)$ at line 18, the execution goes back to line 9 for the third time and the same process goes on over and over again. Thus the program is an infinite loop of computing and printing the factorial of 5.

```
1   Continuation x;
2
3   int fact(int n)
4   {
5     if (n == 0) {
6       continuation c {
7         x = c;
8       }
9       return 1;
10    }
11    else
12      return n*fact(n-1);
13  }
14
15  int main()
16  {
17    cout << fact(5) << endl;
18    call(x);
19  }
```

Fig. 3.1: an Example of Factorial with First Class Continuations

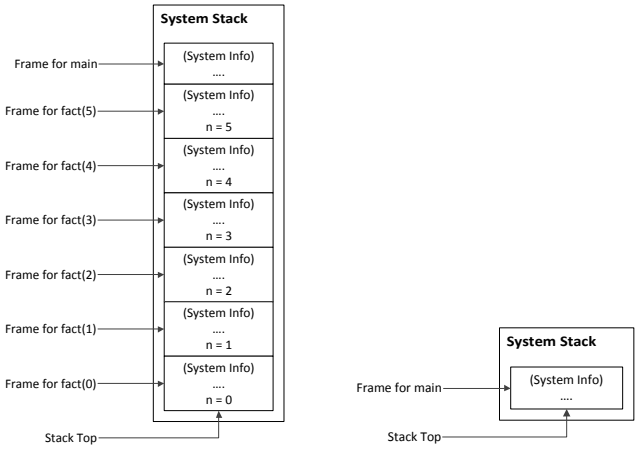
The following is what the execution looks like with a strategy of copying and restoring the system stack.

1. At first $fact(5)$ calls $fact$ recursively. When $n = 0$ the system stack is shown in Figure 3.2a. The captured continuation c snapshots the stack and keeps an

instruction pointer pointing to the instruction after the continuation statement (line 9). Shown in Figure 3.3.

2. When $call(x)$ is reached, the system stack is shown in Figure 3.2b. Frames for factorial functions have been popped since $fact(5)$ returned. The call statement restores the system stack state captured in c. And the program jumps to line 9. The system stack looks exactly like Figure 3.2a. And the factorial of 5 is returned and printed again.

3. The same process goes over again when $call(x)$ is reached. It is an infinite loop of printing factorial of 5.



(a) System Stack at Continuation Statement

(b) System Stack at Continuation Call

Fig. 3.2: System Stack in the Factorial Example

Unfortunately, for security reasons most systems do not allow a direct manipulation of the system stack. Moreover, function calls and returns leave platform dependent

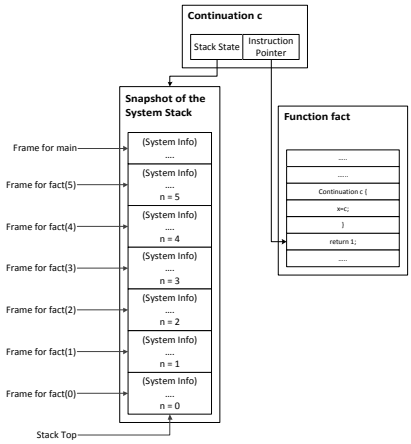


Fig. 3.3: Continuation in the Factorial Example

information on the system stack. Even if we use our own run-time structure rather than the system stack, we cannot capture the execution state without saving and restoring the system stack if function calls and returns are used.

3.2 Stack Based Continuations in Comet

Comet is an imperative language that supports first class continuations. It uses GNU Lightning [6] which has a MIPS like virtual instruction set. The instruction set does not include stack copying instructions. Which means direct copying of the system stack cannot be implemented in a portable way. We have to resort to pure assembly code for stack copying which differs among architectures.

Another way is to avoid the system stack completely. Comet moves all user defined information out of the system stack into the Comet stack. A continuation copies and restores the Comet stack instead, which is enough if at any point no execution state information is on the system stack. The function call and return instructions must

be avoided since they push and pop data on the system stack. Thus COMET function calls and returns are actually jumps with pushing and popping of the COMET stack. The whole program becomes a large function with jumps across its body.

As an example, Figure 3.4 shows the differences between conventional programming languages and COMET of the factorial program in Figure 2.2. In COMET a call to $fact$ does the following:

1. Push onto the COMET stack the data and the 'ret' label that tells the callee where to return (in this example ret = '$next$').

2. Jump to label $fact$.

A return in $fact$ does the following:

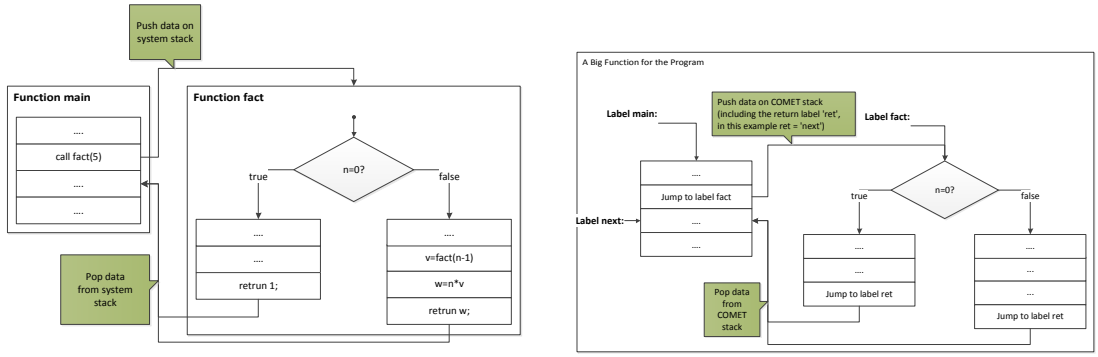
1. Read the 'ret' label from the COMET stack.

2. Pop out the data from the COMET stack (including the 'ret' label).

3. Jump to label ret.

Now the system stack is only used by library function calls. No other information on the system stack will be used at any point of the program execution. A call to continuation restores the COMET stack and goes to the instruction after the continuation statement.

To optimize the saving and restoring of the stack, COMET adopts an incremental scheme. When COMET saves the stack for the first time, it copies the entire stack. The full copy is known as the reference stack. A subsequent saving computes the



(a) Function in Conventional Imperative Language **(b)** Function in COMET

Fig. 3.4: Comparison of Functions in Conventional Imperative Language and COMET

differences between the current stack and the reference stack, and saves the differences in a incremental snapshot (along with the pointer to the reference stack). When an incremental snapshot is restored, the reference stack is first restored then the differences are restored. This incremental scheme is a space saving measurement.

3.3 Stack Based Continuations in AMIBE

AMIBE uses the LLVM compiler infrastructure. LLVM is a modern compiler infrastructure that supports a variety of optimizations. Like GNU Lightning it does not support explicit manipulation of the system stack. We want to get rid of the system stack and move the entire execution state to the AMIBE stack.

Now we face the same problem: we need to avoid the function call and return instructions in LLVM. Suppose we use the same technology in GNU Lightning that turns the whole program into one big function. It would not be practical to optimize the huge function in LLVM because the complexity of control flow optimizations would be

unacceptable. To exploit the LLVM optimizations it is necessary to keep real functions and function calls. We face two contradictory requirements:

1. Function calls and returns should be avoided because they leave part of the execution state on the system stack, which LLVM cannot manipulate explicitly.

2. Real functions and function calls are needed in LLVM to keep the computation cost of optimizations reasonable.

Compiling with Continuation Passing Style(CPS) gets us out of this dilemma. A function compiled with CPS is passed a continuation which represents the future execution. When its computation ends, instead of returning it calls the continuation to perform the future computations. In CPS a function calls another function to continue the execution and never returns. All function calls are tail calls.

As an example Figure 3.5 shows the AMIBE function for the factorial program. A call to $fact$ does the following:

1. Push onto the AMIBE stack the data and the 'ret' continuation function that should be called when callee returns (in this example ret = '$main$').

2. Call the function $fact$

A return in $fact$ does the following:

1. Read the 'ret' continuation function from the AMIBE stack.

2. Pop out the data from the AMIBE stack (including 'ret').

3. Call the function ret.

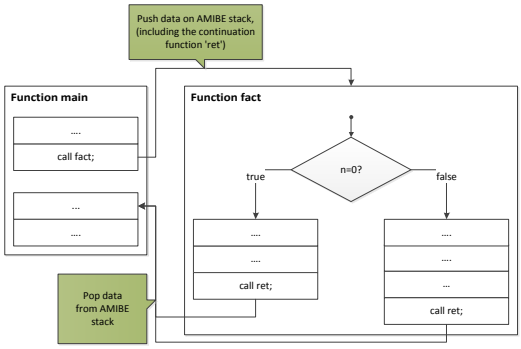


Fig. 3.5: Function in AMIBE

The information on the system stack is never reused since functions never return(i.e. information on the system stack is not part of the execution state). There is no need for first class continuations to save and restore the system stack. Saving and restoring the AMIBE stack is enough.

Unlike functional programming languages compiled with continuations, compiling with CPS in AMIBE does not require a full-scale CPS transformation. Only function calls and returns are compiled in CPS, which is enough to correctly implement first class continuations. This partial CPS compilation is done by first generating the LLVM three address code, then going through a CPS transformation. The design of the CPS transformation is discussed in Chapter 4. The implementation details are discussed in Chapter 8.

Chapter 4

Continuation Passing Style in AMIBE

4.1 Introduction to CPS in AMIBE

Since in LLVM there is no explicit way to manipulate the system stack, then the execution state need to be moved out of the system stack into the AMIBE stack – a run-time structure that can be manipulated by the AMIBE compiler. The system stack should not contain any state information related to the execution, which can be achieved by compiling with Continuation Passing Style(CPS). In CPS functions never return, therefore the state on the system stack is never reused and does not need to be saved by continuations.

Unlike functional programming languages based on CPS where programs are compiled into a pure CPS IR, in AMIBE only function calls and returns are compiled with CPS. Basic operations (e.g. branching, arithmetical operations) are compiled into three address codes that are closely related to LLVM IR. Reasons to do this partial CPS compiling rather than a full-scale CPS compiling are given below:

1. Compiling the function calls and returns with CPS is enough to implement first class continuations.

2. AMIBE IR in three address code that is closely related to LLVM IR makes it easy to translate AMIBE IR into LLVM IR and exploit the optimizations provided by LLVM.

The partial CPS compiling consists of two steps:

1. Compile the program into AMIBE IR(three address code) with real function calls and returns.

2. Do a CPS transformation on the functions. After the transformation, a function has multiple entry points for both ordinary function calls and continuation calls. The call and return instructions are in CPS form. Details are discussed in the following sections.

The AMIBE IR code generation is discussed in Chapter 7. The discussion on the CPS transformation in this chapter is based on the assumption that we already compiled AMIBE program into IR.

4.2 AMIBE Function Before the CPS Transformation

In AMIBE a running function consists of the function body(a list of basic blocks) and its frame on the AMIBE stack. A basic block is a list of IR instructions that can only be entered from the head and exit from the tail, without any branching in the middle, shown in Figure 4.1.

Function calls and returns only appear on the exit of basic blocks. The only basic block following a function call is called the continuation block to that call. There is no successor to a basic block ending with a return, see the illustration in Figure 4.2.

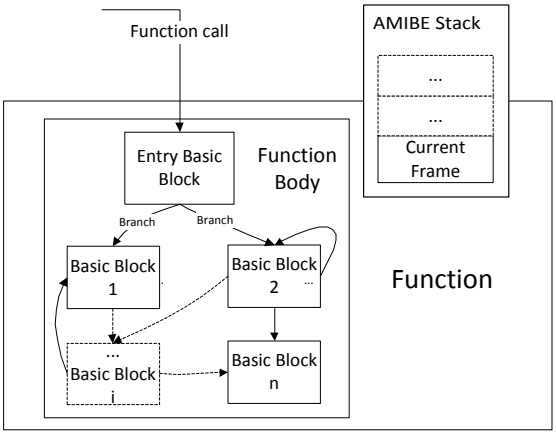
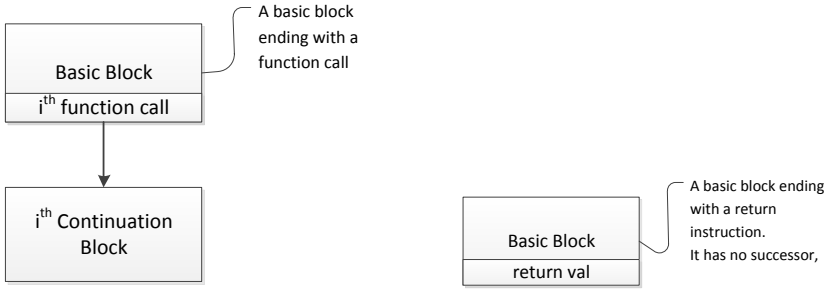


Fig. 4.1: Function in AMIBE

Without loss of generality, in a function f every function call is assigned an unique number $i(1 \leq i \leq n$, where n is the total number of function calls in $f)$. A function call with assigned number i is called the i^{th} function call. Its successor is called the i^{th} continuation block.



(a) function call **(b)** return

Fig. 4.2: Function Call and Return Basic Blocks in AMIBE

As an example, the factorial function in Figure 4.3 before the CPS transformation

is shown in Figure 4.4. There is only 1 function call in $fact$, we call $t2 = fact(t1)$ the 1^{st} function call and the basic block following it the 1^{st} continuation block.

```
1   int fact(int n)
2   {
3     if (n==0)
4       return 1;
5     else
6       return n*fact(n-1);
7   }
```

Fig. 4.3: the Factorial Function in AMIBE

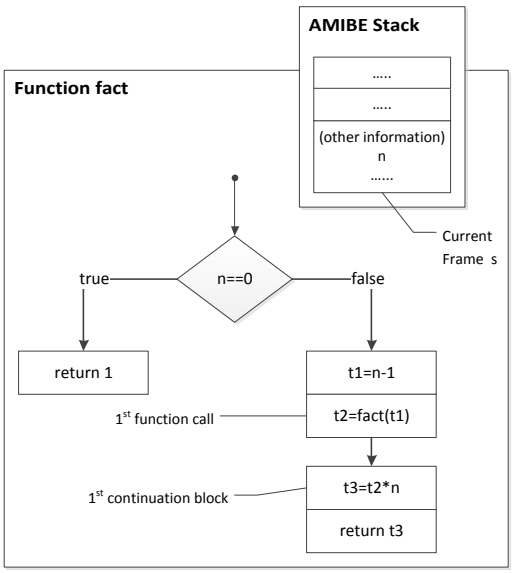


Fig. 4.4: Factorial Function Before the CPS Transformation

4.3 Continuations in CPS

A function compiled with CPS is passed a continuation when it is called. When the function completes its job, rather than returning it calls the continuation to resume

the execution. Thus for a function call the caller needs to create a continuation that represents the future of the execution and passes it as an extra argument to the callee.

The continuation to a function call is the future of the function call which can be determined by:

1. the context in the current function

2. pointer to the instruction following the function call

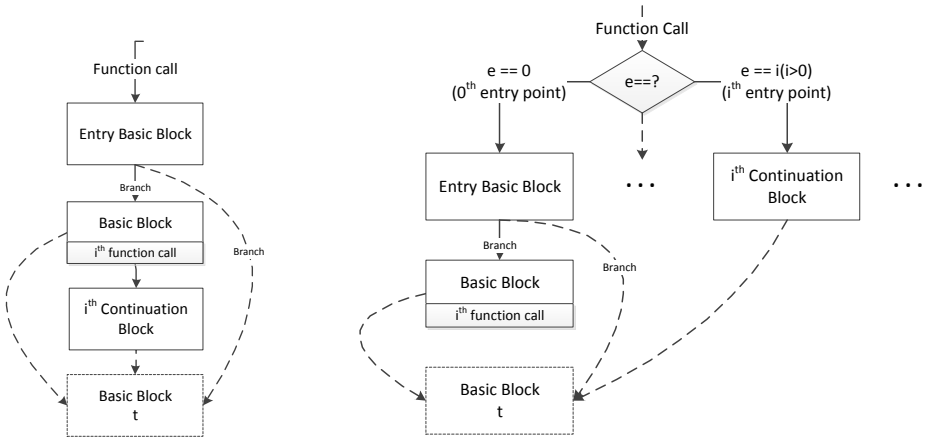
Notice that the continuation here refers to the continuation in CPS. It is a concept used in compiling with CPS, not a first class object in the AMIBE language.

4.4 CPS Transformation for Functions

To represent the continuation of a function call, the CPS transformation adds an extra entry point for every function call in a function. If there are n function calls in a function f, there are $n+1$ entry points after the CPS transformation. The entry point 0 is for the ordinary function call. Entry points 1 to n are for continuation calls. Each corresponds to the continuation to a function call in f. We call the entry point corresponding to the i^{th} function call the i^{th} entry point.

The CPS transformation disconnects the i^{th} continuation block from i^{th} function call($1 \leq i \leq n$) and treats it as the entry basic block to entry point i. When entering from entry point 0, the execution starts an ordinary function call. When entering from entry point i, the execution jumps to the i^{th} continuation block. A multi-entry points function after the CPS transformation is show in Figure 4.5.

For instance, there are 1 function call in the factorial example. After the CPS



(a) Before the CPS transformation

(b) After the CPS transformation

Fig. 4.5: Function Before and After the CPS transformation

transformation there are 2 entry points. The 0^{th} entry point is for the ordinary function call. The 1^{st} entry point points to the 1^{st} continuation block, shown in Figure 4.6.

With multiple entry points in place, the continuation to the i^{th} function call in the function f can now be represented as

1. the function pointer to f
2. the entry point number i
3. the current frame s

For convenience we represent such a continuation as a triple $< f, i, s >$. For instance, the continuation to the 1^{st} function call in $fact$ is $< fact, 1, s >$.

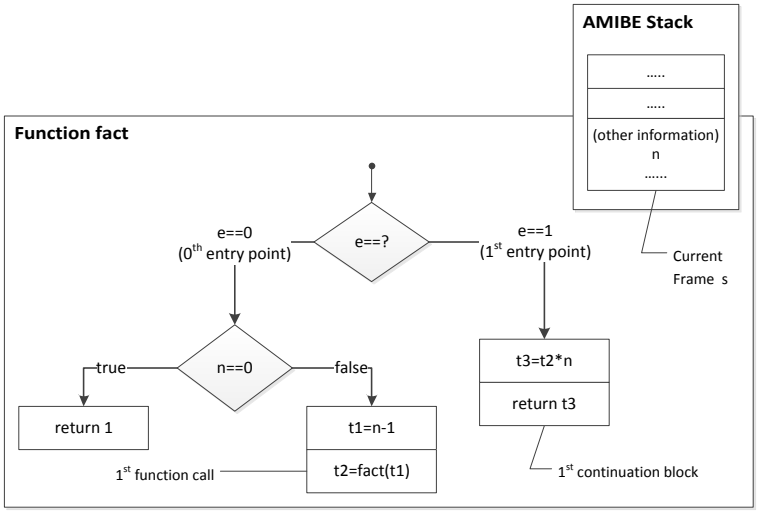


Fig. 4.6: Factorial Function After the CPS Transformation

4.5 CPS Transformation for Function Calls and Returns

The i^{th} function call ($g(a_1, ..., a_m)$) in function f is transformed into CPS form as follows:

1. Creates a continuation $< f, i, s >$ where s is the current frame.

2. Pass two extra arguments to the function call: the entry point number $e = 0$ (which means this is an ordinary function call), and the continuation $< f, i, s >$.

   The call instruction becomes:

$$g(a_1, ..., a_m) \Longrightarrow g(0, < f, i, s >, a_1, ..., a_m) \tag{4.1}$$

   The return instruction (return v;) is transformed in CPS as follows:

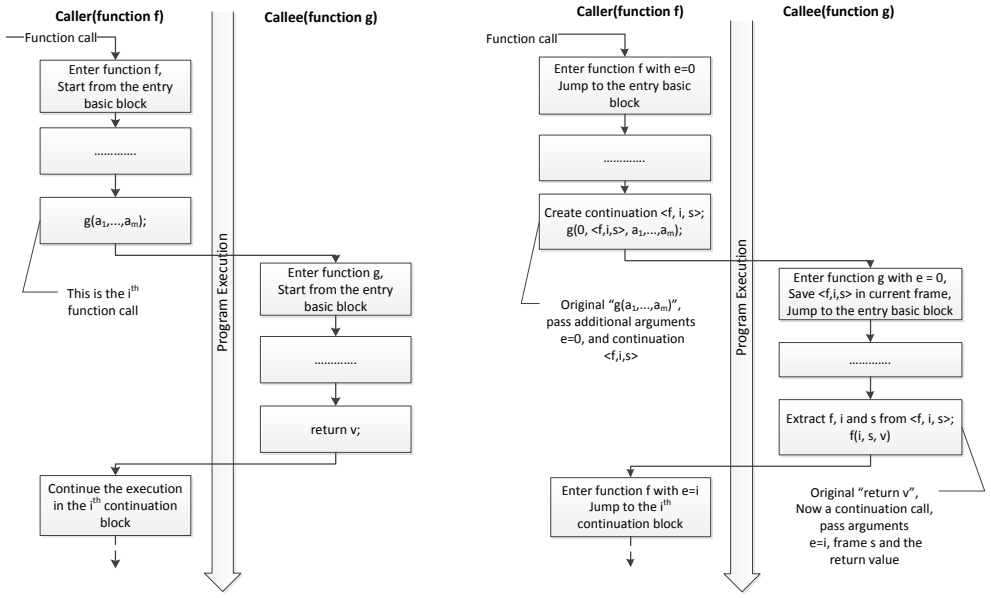
1. Extracts f, i, s from the continuation $< f, i, s >$.

2. Call function f with i, s and v as arguments.

The return instruction becomes:

$$\text{return } v; \Longrightarrow f(i, s, v); \tag{4.2}$$

The control flow of function call and return before and after the CPS transformation is show in figure 4.7.



(a) before the CPS transformation **(b)** after the CPS transformation

Fig. 4.7: Function Call and Return Before and After the CPS transformation

As an example, the factorial function after transforming calls and returns is shown in Figure 4.8.

In the factorial example assume $fact(1)$ is called (which is CPS transformed into $fact(0, k, 1)$ where $k =< f_k, i_k, s_k >$ is a continuation). It jumps to entry point 0. The $fact$ function knows it is an ordinary function call since $e == 0$. The 1^{st} function call

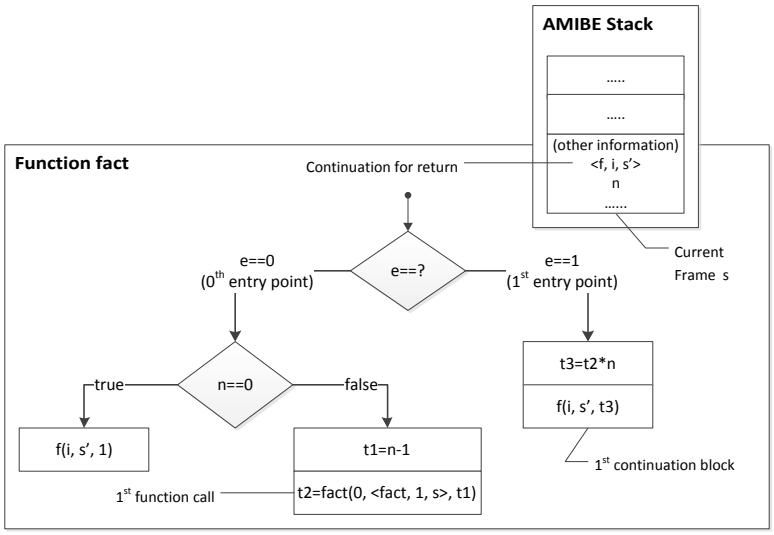


Fig. 4.8: Factorial Function after Transforming Call and Return

$(t2 = fact(t1))$ is reached first since $n == 1$, which creates a continuation $< fact, 1, s >$. The call instruction is transformed into the following:

$$fact(t1) \Longrightarrow fact(0, < fact, 1, s >, t1) \tag{4.3}$$

When the callee takes over, it stores the continuation $< fact, 1, s >$ as $< f, i, s' >$ in its frame and jumps to the entry point 0. The return instruction (return 1;) is reached since $n == 0$. It extracts the function pointer f, the entry point i and the frame s' from the continuation $< f, i, s' >$. The function pointer f is called with the entry point i, the frame s' and the return value 1 as arguments. That is: rather than returning it calls the caller f from entry point i with the return value and the frame s' as arguments. The return instruction becomes:

$$\text{return } 1 \Longrightarrow f(i, s', 1) \tag{4.4}$$

In this example $f = fact$ and $i = 1$. Now the execution returns to the caller. It enters the entry point 1 and jumps to the 1^{st} continuation block with the current frame set to s'. With the frame restored and $t2$ set to the return value 1, the program correctly continue the future of $(t2 = fact(t1))$. It computes $t3 = t2 * n = 1 * 1 = 1$ and reaches another return instruction (return $t3$;), which is transformed into the following:

$$\text{return } t3 \Longrightarrow f(i, s', t3) \tag{4.5}$$

This time $f = f_k$, $i = i_k$ and $s' = s_k$. That is: the program calls continuation $k =< f_k, i_k, s_k >$ with $t3$ as the return value, where $t3$ is exactly the result of $fact(1)$.

4.6 First Class Continuations in AMIBE

This section describes the two statements for first class continuations. The first subsection describes continuation statements which create first class continuation objects. The second subsection describes calls to continuation objects.

4.6.1 First Class Continuation Statements

A first class continuation is created by a continuation statement, shown below:

```
1  ... // state to be saved
2  continuation c {
3     ... // capture block
4  }
5  ... // resume point
```

Fig. 4.9: Continuation Statement in AMIBE

When the continuation is called the execution starts at the resume point with the execution state restored to its value prior to the capture.

The continuation statement in AMIBE is explained by using *call/cc* in *Scheme* which stands for "call-with-current-continuation" [24] [7]. *call/cc* takes as input a function f which takes one argument(the current continuation). It applies f to the current continuation. The continuation statement

```
1  continuation c { BLOCK }
```

is semantically the same as the *call/cc* as follows:

```
1  call/cc((lambda (Continuation c) BLOCK))
```

The lambda function (lambda (Continuation c) BLOCK) is applied to the current first class continuation t, semantically the same as the code in Figure 4.10:

```
1  ...
2  void capture(Continuation c) { BLOCK }
3  t = create_first_continuation ;
4  capture(t);
5  ...
```

Fig. 4.10: AMIBE Code for First Class Continuation Statement in AMIBE

Take the factorial continuation program in Figure 3.1 (repeated here in Figure 4.11) as an example. The implementation with *call/cc* is shown in Figure 4.12. The implementation in AMIBE is shown in Figure 4.13.

The fact function in Figure 4.13 after the CPS transformation is shown in Figure 4.16. A continuation $< fact, 2, s >$ for the function call at line 10 in Figure 4.13 is created (assume it is the 2^{nd} function call). The $create\_first\_class\_continuation$ instruction does two things:

1. gets a snapshot S of the AMIBE stack

```
1   Continuation x;
2
3   int fact(int n)
4   {
5     if (n == 0) {
6       continuation c {
7         x = c;
8       }
9       return 1;
10    }
11    else
12      return n*fact(n-1);
13  }
```

Fig. 4.11: Factorial Continuation Function

```
1   Continuation x;
2
3   int fact(int n)
4   {
5     if (n == 0) {
6       call/cc((lambda (Continuation c) (x=c)));
7       return 1;
8     }
9     else
10      return n*fact(n-1);
11  }
```

Fig. 4.12: Factorial Continuation Function Implemented with *call/cc*

2. creates a first class continuation $t =< fact, 2, s_c >$ where s_c points to the current frame in the snapshot S.

4.6.2 First Class Continuation Calls

When a first class continuation $t =< f, i, s_c >$ is called, it should go to the resume point in Figure 4.9 with the AMIBE stack state restored. The continuation call statement in Figure 4.14 is transformed into the code in Figure 4.15.

The *restore_state* function copies the snapshot back to the AMIBE stack and

```
1   Continuation x;
2
3   int fact(int n)
4   {
5     if (n == 0) {
6       void capture(Continuation c) {
7         x = c;
8       }
9       t = create_first_continuation ;
10      capture(t);
11      return 1;
12    }
13    else
14      return n*fact(n-1);
15  }
```

Fig. 4.13: Factorial Continuation Function Implemented in AMIBE

```
1   ...
2   call(t); // call the continuation t = <f, i, s_c>
3   ...
```

Fig. 4.14: Continuation Call in AMIBE

```
1   ...
2   s = restore_state(s_c);
3   f(i, s);
4   ...
```

Fig. 4.15: Implementation of Continuation Call in AMIBE

returns the current frame s on the restored stack. Then the execution enters f from the i^{th} entry point and jumps to the i^{th} continuation block(i.e. the instruction at the resume point) with the current frame set to s.

In the factorial continuation example in Figure 3.1, the *main* function calls continuation $x(x =< fact, 2, s_c >)$. With the AMIBE stack snapshot in Figure 4.16 restored, it calls $fact(2, s)$(where s is the current frame on the restored stack) and enters $fact$

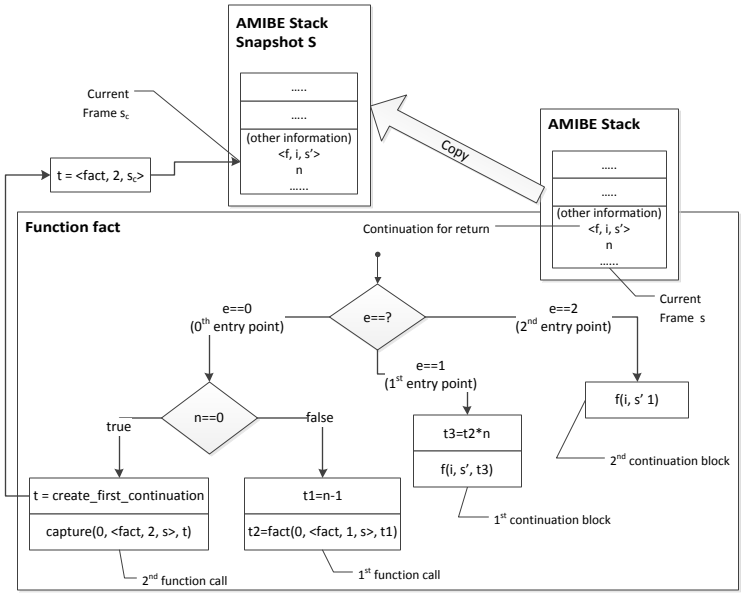


Fig. 4.16: Factorial Continuation Function after CPS Transformation

function from the 2^{nd} entry point with the current frame set to s. The execution starts from the 2^{nd} continuation block which is exactly the instructions after the continuation statement in $fact$.

We can see that compiling function calls and returns with CPS makes first class continuation implementation easy:

1. the continuation statement is similar to a function call after the CPS transformation, except it needs to create a first class continuation before the call.

2. the continuation call statement is similar to a function return after the CPS transformation, except it needs to restore the execution state before the call.

4.7 Consequences of the CPS Transformation

From the discussions above we can see that after the CPS transformation the execution always goes forward by calling functions until it reaches the end of the program or an exception is raised. Although function calls causes the system stack to grow, information on the system stack is never reused as functions never return, which means a first class continuation does not need to save the system stack information. All the execution state related information is on the AMIBE stack. That's why in our discussion for implementing first class continuations, we never mentioned the system stack.

Although the information on the system stack is irrelevant to the correctness of programs, they might cause the system stack overflow as program calls functions infinitely. That's where LLVM tail call optimization plays a crucial role. After the CPS transformation, function calls(except the library function calls) only appear on the exit of basic blocks. Every function call is tail call. The LLVM optimizer provides a tail call optimization to transform tail calls into jumps. Function calls are replaced by jumps after the optimization. The system stack will never grow on function calls (except for library calls). Details are introduced in Chapter 9.

After the CPS transformation, a function call may be either an ordinary function call(when $e = 0$) or a continuation call(when $e > 0$). And the number of arguments passed to a function differs. The technique to pass variant arguments are discussed in Chapter 7.

Chapter 5

Compiler Organization

5.1 Introduction

This chapter give a brief overview of the organization of the AMIBE compiler. The comprehensive description of the classic techniques used for compilers can be found in many compiler books [3] [5],.

The AMIBE compiler takes a source program as input and outputs the LLVM IR in CPS form. It consists of the following passes:

1. Compiler front-end
2. AMIBE IR code generation
3. CPS transformation
4. LLVM code generation and optimizations

Each pass takes the output of the last pass as input. And the final result is LLVM IR in CPS form. The LLVM IR is then compiled into assembly code and executable on specific platforms by the LLVM compiler and optimizer (as shown in Figure 5.1). The AMIBE compiler is primarily written in C++ (about 20,000 lines of C++ code).

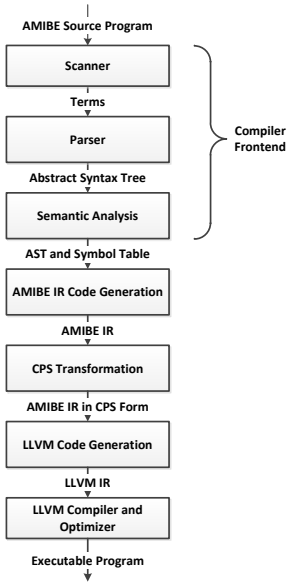


Fig. 5.1: The AMIBE Compiler

5.2 Compiler Front-end

The compiler front-end includes the scanner, parser and semantic analysis passes. A source program is scanned, parsed and analyzed in the front-end. The output is an abstract syntax tree(AST) that is semantically correct and a symbol table containing the types, variables and functions objects in the program. The implementation of the compiler front-end is discussed in Chapter 6.

5.3 AMIBE IR Code Generation

The intermediate representation(IR) of AMIBE is a set of virtual instructions which closely relate to the LLVM IR. The AMIBE IR is similar to most three address instructions for conventional imperative languages [3] [5], with extra instructions for first

class continuations and the CPS transformation. The IR code generation pass takes an AST and a symbol table as input and outputs the AMIBE IR with real function calls and returns. The implementation of AMIBE IR code generation is discussed in Chapter 7.

5.4 CPS Transformation

The CPS transformation pass transforms non-CPS AMIBE IR into one in CPS form. After the CPS Transformation, functions calls never return. Note that this is not a whole program CPS transformation as shown in [4]. Instead, it offers a novel partial transformation that only focuses on eliminating traditional function calls and operates correctly in a stack-based implementation for languages with destructive primitives (e.g., assignments). The implementation of the CPS transformation is discussed in Chapter 8.

5.5 LLVM Code Generation and Optimizations

The LLVM code generation pass translates the AMIBE IR in CPS form into LLVM IR. The tail-call optimization replaces function calls by jumps. Their implementation is developed in Chapter 9.

Chapter 6

Compiler Front-End

6.1 Introduction

The front-end of the AMIBE compiler takes a source program as input and outputs an Abstract Syntax Tree and a symbol table. The source program first goes through the scanner, then the parser and finally the semantic analysis.

6.2 Scanner and Parser

The scanner takes a source program as input and outputs tokens for the parser. It is implemented by using $flex$ [2]. The parser takes tokens generated by the scanner and outputs an AST of the source program. It is implemented by using $Bison$ [1].

6.3 Abstract Syntax Tree

A partial class hierarchy of the AST is shown in Figure 6.1. The AST classes are listed in Table 6.1. The $ASTNode$ class is the root class of the hierarchy. $Program$ stands for an AMIBE program that contains a list of function definitions or declarations ($FuncDef$ or $ExtFuncDecl$ objects). $Stmt$ is the parent class for all statements classes. $Expr$ is the parent class for all expression classes.

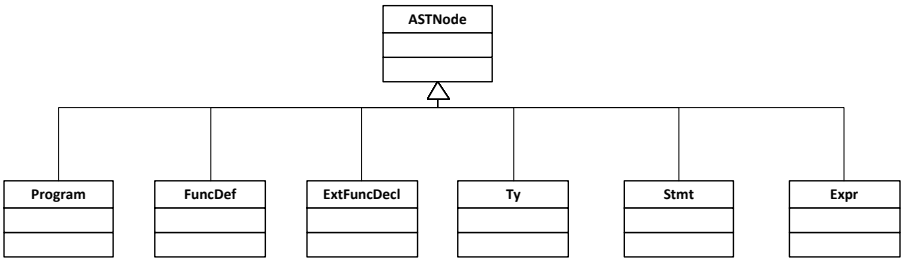


Fig. 6.1: AMIBE Abstract Syntax Tree Class Hierarchy

Name	Abstract	Description
ASTNode	Yes	The parent class of all AST classes
Program	No	The AMIBE program containing a list of function definitions or declarations
FuncDef	No	A function definition in the program.
ExtFuncDecl	No	An external function declaration that refers to library functions not defined in the program
Ty	Yes	The parent class for type classes
Stmt	Yes	The parent class for statement classes
Expr	Yes	The parent class for expression classes

Table 6.1: AST Classes

The class hierarchy for statement classes is in Figure 6.2. Details for the statement classes are in Table 6.2. The class hierarchy for expression classes is in Figure 6.3. Details for the expression classes are in Table 6.3.

6.4 Symbol Table

The symbol table stores a list of triples. A triple represents one named object in the program. It has the following format

$$< name, scope, symentry > \tag{6.1}$$

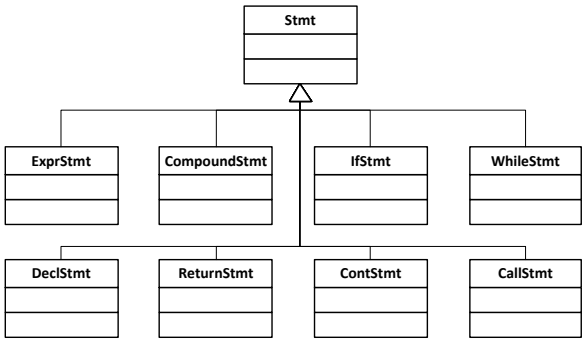


Fig. 6.2: AMIBE Statement Class Hierarchy

Name	**Description**	**Format**
ExprStmt	The expression statement	E;
CompoundStmt	A list of statements	$\{S_1, ..., S_n\}$
IfStmt	The if statement	if (E) S_1 else S_2
WhileStmt	The while statement	while (E) S
DeclStmt	The declaration statement	T $x = E$;
ReturnStmt	The return statement	return E;
ContStmt	The continuation statement	continuation x $\{S\}$
CallStmt	The call to continuation	call (E);

Table 6.2: AMIBE Statement Classes

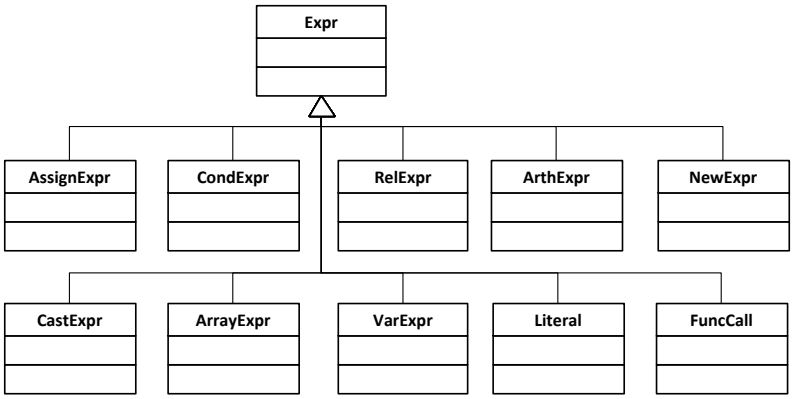


Fig. 6.3: AMIBE Expression Class Hierarchy

Name	Description	Format
$AssignExpr$	The assignment expression	$E_1 = E_2$
$CondExpr$	The conditional expression, including 'not','and', 'or' expressions	$!E_1$, $E_1 \&\& E_2$,$E_1 \| \| E_2$
$RelExpr$	The relational expression, including 'equal', 'not equal', 'less', 'less than' expressions, etc.	$E_1 == E_2$, $E_1 != E_2$, $E_1 < E_2$, $E_1 <= E_2$, etc.
$ArthExpr$	The arithmetical expression class, including 'add', 'subtract', 'multiple', 'divide', 'remainder' expressions, etc.	$E_1 + E_2$, $E_1 - E_2$,$E_1 * E_2$, E_1/E_2, $E_1\%E_2$, etc.
$NewExpr$	New expression.	new $T(E)$
$CastExpr$	Cast between different types.	$(T)(E)$
$ArrayExpr$	Get an array element.	$E_1[E_2]$
$VarExpr$	Variable expression.	x
$Literal$	Integer, real, character literals.	$1, 'c', 2.3333$
$FuncCall$	Function call expression.	$E(E_1, E_2, .., E_m)$

Table 6.3: Expression Classes

name is a string representing the name of the entry. *scope* is an unique integer representing the scope in which the name is defined. The same name cannot appear twice in any given scope, which means the pair $< name, scope >$ uniquely identifies an object.

symentry is the entry for the object represented by $< name, scope >$. The class $SymTabEntry$ is the abstract parent class for all symbol entries. A concrete symbol entry might be one of the following(as shown in Figure 6.4):

1. $TypeEntry$: a type entry,

2. $VarEntry$: a variable entry,

3. $FuncEntry$: a function entry.

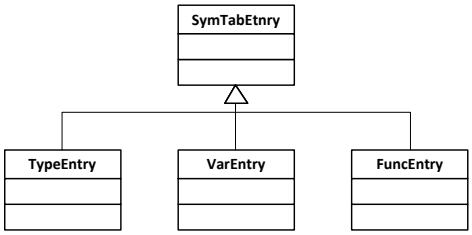


Fig. 6.4: Symbol Entry Class Hierarchy

6.5 Semantic Analysis

The semantic analysis takes an AST as input and outputs the semantically checked AST and a symbol table. The AST goes through the following semantic analysis passes:

1. Creating Named Object Pass. It finds all the named objects in the program including types, variables and functions and creates $TypeEntry$, $VarEntry$ and $FuncEntry$ objects for them. The name, scope, and entries of these objects are stored in a symbol table.

2. Declaration Pass. The types of the objects defined in the declaration statements (variable declarations and function declarations) are created. The symbol table entries are updated accordingly.

3. Name Resolution Pass. The named objects used in expressions and statements are looked up lexically (as AMIBE is a lexically scoped language like C++/Java). Exceptions are raised if any object is used before it is defined.

4. Type Checking Pass. The type correctness of expressions and statements are checked. For instance, only 2 integers and 2 real values can be added in the

addition expression, a function call must pass arguments with exactly the same type as parameters in the function declaration, etc.

These passes are implemented using *Visitor* pattern [12]. An *ASTVisitor* abstract class is defined that provides a *VisitXXX* method for every AST class *XXX*. *ASTNode* class contains an abstract *Accept* method that accepts a *ASTVisitor*. An concrete AST class *XXX* implements the *Accept* method that calls *VisitXXX* method. The class hierarchy of the semantic analysis passes is shown in Figure 6.5. The control flow of semantic check passes is shown in Figure 6.6

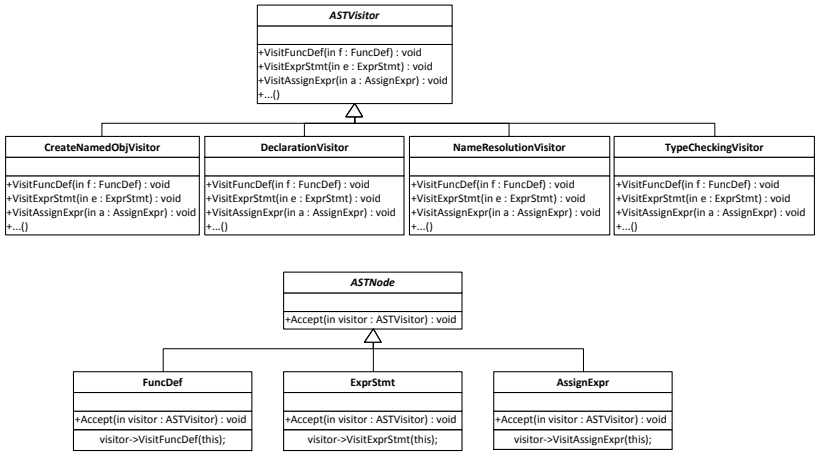


Fig. 6.5: Visitor Pattern in the Semantic Analysis

6.6 The Factorial Continuation Example

Throughout the next few chapters, the factorial continuation example in Figure 2.4 is used to demonstrate how the AMIBE compiler works. The program is repeated in Figure 6.7.

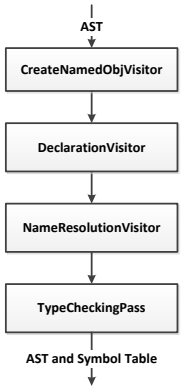


Fig. 6.6: Control Flow of Semantic Passes

```
1   int fact(int n, Continuation[] x)
2   {
3     if (n == 0) {
4       continuation c {
5         x[0] = c;
6       }
7       return 1;
8     }
9     else
10      return n*fact(n-1, x);
11  }
12
13  int main()
14  {
15    Continuation[] x = new Continuation[](1);
16    int d = fact(5, x);
17    call(x[0]);
18  }
```

Fig. 6.7: a Factorial Continuation Example in AMIBE

In the compiler front-end, the program is first scanned and parsed into an AST shown in Figure 6.8 (As the space is limited, it is not the complete AST. For example, the *RelExpr* $n == 0$ has subnodes n and 0 which are not shown in the figure). Since it is semantically correct, the semantic analysis passes output a symbol table as shown

in Table 6.4.

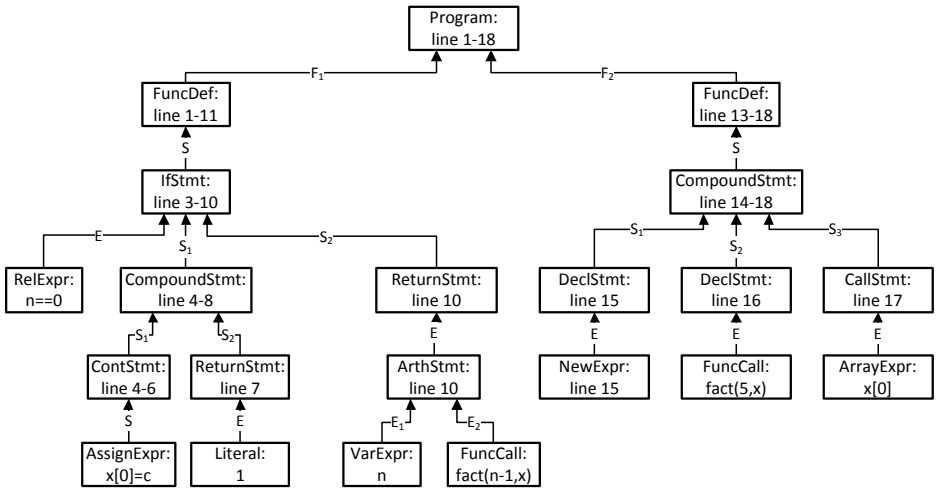


Fig. 6.8: AST for the Factorial Continuation Example

Name	Scope	Symbol Entry
real	0	TypeEntry
Continuation	0	TypeEntry
int	0	TypeEntry
bool	0	TypeEntry
void	0	TypeEntry
char	0	TypeEntry
fact	1	FuncEntry
main	1	FuncEntry
n	2	VarEntry
x	2	VarEntry
c	5	VarEntry
x	7	VarEntry
d	8	VarEntry

Table 6.4: Symbol Table for the Factorial Continuation Example

Chapter 7

AMIBE IR Code Generation

7.1 Introduction

The IR code generation pass translates the AST and the symbol table into AMIBE IR. First the virtual machine for the AMIBE IR (including the values, virtual machine instructions, run-time structures) is developed. Then the translation from the AST to AMIBE IR is developed. The translation is very similar to what a conventional imperative programming language would do [3] [5], except that some instructions for compiling with CPS are generated.

7.2 AMIBE Virtual Machine

The virtual machine consists of the IR value, virtual machine instructions and run-time structures (such as functions, frames and the AMIBE stack).

7.2.1 IR Values

The IR values are objects manipulated by the virtual machine instructions. The class hierarchy of the value classes is shown in Figure 7.1. *Val* class is the parent class of all concrete value classes. Details of the value classes are in Table 7.1.

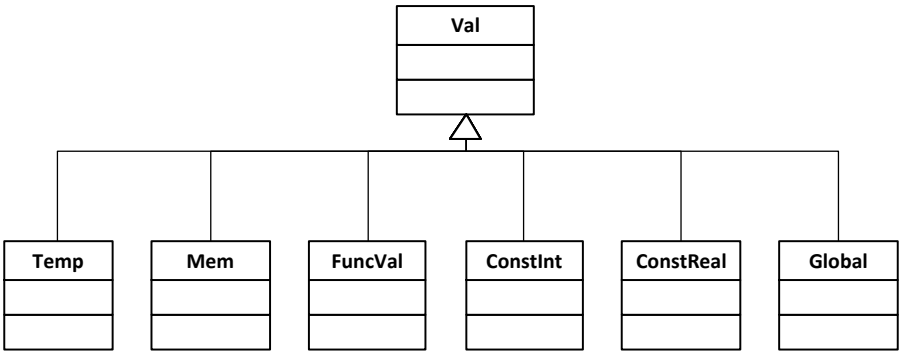


Fig. 7.1: Class Hierarchy of IR Values

Name	**Constructor**	**Description**	**Symbol**
Value	–	the parent class of all concrete value classes	v
Temp	$Temp(id)$	a temporary value with name 't_{id}'	t
Mem	$Mem(v_1, v_2)$	the memory address to v_2^{th} cell of the composite value pointed by v_1 ($v_2 \geq 0$)	m
FuncVal	$FuncVal(f)$	the function f	f
ConstInt	$ConstInt(i)$	an integer constant i	i
ConstReal	$ConstReal(r)$	a real constant r	r
Global	$Global(g)$	a global variable that occupies a region of memory cells	g

Table 7.1: IR Value Classes

Each value occupies one cell (which has a fixed length) in the memory. A basic object such as integer, real, character contains one value(cell). A composite object contains several values(cells). Composite objects are referred by pointers that occupy one cell. Its values can be access by using Mem. For example, if m is a pointer to a composite object c, then $Mem(m, 2)$ is a pointer to the address of the 2^{nd} cell of c. With fixed length cells we do not have to worry about the padding and alignment in the real machine.

7.2.2 Virtual Machine Instructions

A virtual machine instruction takes IR values as inputs and outputs IR values. The instructions of AMIBE IR are three address codes (listed in Table 7.2).

Name	Format	Description
Store	st m, v	store the value v into the memory cell pointed by m
Load	v = ld m	load the value in m into v
New	m = new i	allocate i consecutive cells from the memory and m points the head of the cells
ICmp/RCmp	$v = v_1\ OP_c\ v_2$	compares the integer/real values, put the result in v, OP_c can be ==,! =, >, ≥, < or ≤
IAdd/RAdd	$v = v_1 + v_2$	add two integer/real values, put the result in v
ISub/RSub	$v = v_1 - v_2$	subtract two integer/real values, put the result in v
IMul/RMul	$v = v_1 * v_2$	multiply two integer/real values, put the result in v
IDiv/RDiv	$v = v_1$ / v_2	divide two integer/real values, put the result in v
IMod/RMod	$v = v_1$ % v_2	get the remainder of two integer/real values, put the result in v
Label	l:	a label l
Br	br l	unconditional jump to the label l
CBr	cbr v l_1 l_2	conditional jump, if v is not zero, jump to label l_1, otherwise jump to l_2
Halt	halt(msg)	end the execution and print out the message *msg*
Call	call $f(v_1, .., v_m)$	call a function f with arguments$(v_1, .., v_m)$
CPSCall	[CPS] call $f(v_1, .., v_m)$	a function call before the CPS transformation
Ret	ret v	return a value v

Table 7.2: Virtual Machine Instruction Classes

Most of the instructions are similar to instructions in conventional imperative languages. The instructions related to compiling with CPS are *Call*, *CPSCall*, *Ret*. A *Call* is a plain function call. A *CPSCall* is a function call that needs to be transformed into a plain call by the CPS transformation. Before the CPS transformation, every call to the user defined function is a *CPSCall*. *Ret* also needs to be transformed into plain calls by the CPS transformation. Details are discussed in Chapter 8.

7.2.3 Continuations

A continuation is a triple $< f, i, s >$, where f is a function pointer, i is a entry point number and s is a frame pointer(as introduced in Section 4.4). In the abstract machine, a continuation is a composite object of size 3, as shown in Figure 7.2.

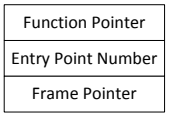


Fig. 7.2: A Continuation

7.2.4 Frames

A frame contains data that can be accessed by a running function. A frame is a composite object, as shown in Figure 7.3. Each frame has a unique frame index. The continuation cell is a pointer to a continuation that should be called when the function returns. AMIBE allows nested functions (as shown in Figure 4.13). The access link cell is a pointer to the frame of its lexical parent function. A function object is a composite variable of size 2 which contains a function pointer and the pointer to the frame of its parent function. A frame also contains local and temporary variables.

7.2.5 AMIBE Stack

The AMIBE stack is a FIFO structure, the frames and continuations are allocated on the stack on function calls and deallocated from the stack on returns. It is discussed in detail in Chapter 8.

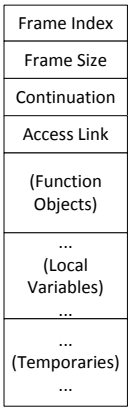


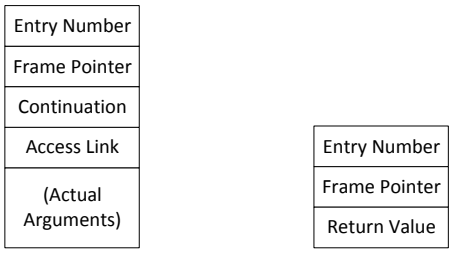
Fig. 7.3: A Frame

7.2.6 Functions

A running function consists of basic blocks that contain virtual machine instructions and a frame.

7.2.7 Argument Passing Store

As discussed in Chapter 4, a AMIBE function takes variant number of arguments after the CPS transformation. An argument passing store is used to pass variant number of arguments to a function. Arguments are written into the argument passing store by the caller and read by the callee. The structure of the argument passing store is in Figure 7.4. On an ordinary function call, the arguments to be passed are the entry point number, continuation, access link and actual arguments. On a continuation call, the arguments to be passed are the entry point number, the frame pointer and the return value (if any).



(a) On an Ordinary Function Call

(b) On a Continuation Call

Fig. 7.4: The Argument Passing Store

7.2.8 Library Functions

Table 7.3 lists the library functions used in the IR code generation and the CPS transformation.

Function	**Description**
$t = imalloc(i)$	allocate i cells and return the pointer to the head of the cells
$t = create\_frame(t_1, t_2, t_3)$	create a frame of size t_1. t_2 is the continuation and t_3 is the access link
$t = create\_cont(f, i, s)$	create a continuation $< f, i, s >$ and return the pointer to the continuation
$delete\_frame(t)$	delete the frame t (and the continuation in t).
$t_1 = create\_first\_continuation(t_2)$	create a first class continuation object t_1 from a continuation t_2
$call\_cont(t)$	call the first class continuation object t.
$print\_err\_msg(t)$	print the error message in t.

Table 7.3: Library Functions

7.3 IR Code Generation from AST

The IR code generation is guided by Small-step Operational Semantics (SOS) developed in Chapter 2. The code generation of AST expressions, statements and functions is discussed. We focus on the code generation related to CPS and first class continuations. Classic code generation technologies can be found in many compiler books [3] [5].

7.3.1 IR Code Generation Equations

The code generation equations of AST expressions, statements and functions are defined as follows:

1. The result of translating an $Expr$ object E is a list of virtual instructions I and a $Temp$ object t representing the value of the expression. The code generation of E is in the Equation 7.1.

$$E \Longrightarrow_e < I, t > \tag{7.1}$$

2. Equation 7.2 represents the code generation of the L-Value of E (where I is a list of instructions and m is a Mem object pointing to the address of the value of E).

$$E \Longrightarrow_l < I, m > \tag{7.2}$$

3. The result of translating a $Stmt$ object S is a list of virtual instructions I. The code generation of S is represented in the Equation 7.3.

$$S \Longrightarrow_s I \tag{7.3}$$

4. The result of translating a $FuncDef$ object F is a function fn in IR. The code generation of functions is represented in the Equation 7.4.

$$F \Longrightarrow_f fn \tag{7.4}$$

7.3.2 Code Generation of Expressions and Statements

The translation of some AST expressions are listed below:

Example 7.3.1. *The translation of a $VarExpr$ (x) is in Equation 7.5 and 7.6 (where p is the frame pointer and i is the offset of the variable x in the frame):*

$$\frac{}{x \Longrightarrow_l <, Mem(p,i)>} \tag{7.5}$$

$$\frac{x \Longrightarrow_l <, m>}{x \Longrightarrow_e < [t = ld\ m], t>} \tag{7.6}$$

Example 7.3.2. *The translation of a $ArthExpr$ $(E_1 + E_2)$ is in Equation 7.7:*

$$\frac{E_1 \Longrightarrow_e < I_1, t_1 >, E_2 \Longrightarrow_e < I_2, t_2 >}{E_1 + E_2 \Longrightarrow_e < [I_1, I_2, t = t_1 + t_2], t >} \tag{7.7}$$

Example 7.3.3. *The translation of a $RelExpr$ $(E_1 < E_2)$ is in Equation 7.8:*

$$\frac{E_1 \Longrightarrow_e < I_1, t_1 >, E_2 \Longrightarrow_e < I_2, t_2 >}{E_1 < E_2 \Longrightarrow_e < [I_1, I_2, t = t_1 < t_2], t >} \tag{7.8}$$

Example 7.3.4. *The translation of an $ArrayExpr$ $(E_1[E_2])$ is in Equation 7.9 and 7.10:*

$$\frac{E_2 \Longrightarrow_e < I_2, t_2 >, E_1 \Longrightarrow_l < I_1, m >}{E_1[E_2] \Longrightarrow_l < [I_2, I_1], Mem(m, t_2) >} \tag{7.9}$$

$$\frac{E_1[E_2] \Longrightarrow_l < I, m >}{E_1[E_2] \Longrightarrow_e < [I, t = ld\ m], t >} \tag{7.10}$$

The translation of some AST statements are listed below:

Example 7.3.5. *The translation of an ExprStmt (E;) is in Equation 7.11:*

$$\frac{E \Longrightarrow_e < I, t >}{E; \Longrightarrow_s I} \tag{7.11}$$

Example 7.3.6. *The translation of a CompoundStmt ($\{S_1, .., S_n\}$) is in Equation 7.12:*

$$\frac{S_1 \Longrightarrow_s I_1, ..., S_n \Longrightarrow_s I_n}{\{S_1, .., S_n\} \Longrightarrow_s [I_1, ..., I_n]} \tag{7.12}$$

The code generation of the control-flow expressions and statements (such as $CondExpr$, $IfStmt$ and $WhileStmt$) is well-known in many compiler books [3] [5]. We are not going to repeat the details here. The interesting part is the translation of expressions and statements that are related to CPS and first class continuations, as follows:

Example 7.3.7. *The translation of a FuncCall expression ($E(E_1, ..., E_m)$) is in Equation 7.13:*

$$\frac{E_1 \Longrightarrow_e < I_1, t_1 >, ..., E_m \Longrightarrow_e < I_m, t_m >, E \Longrightarrow_e < I, f >}{E(E_1, ..., E_m) \Longrightarrow_e < [I_1, ..., I_m, I, t = [CPS]\ call\ f(t_1, ..., t_m)], t >} \tag{7.13}$$

The CPSCall instruction will be replaced by a plain call instruction in the CPS transformation.

Example 7.3.8. *The translation of a ReturnStmt statement (return E;) is in Equation 7.14:*

$$\frac{E \Longrightarrow_e < I, t >}{return\ E; \Longrightarrow_s [I, ret\ t]} \tag{7.14}$$

The Ret instruction will be replaced by a plain call instruction in the CPS transformation.

Example 7.3.9. *The translation of ContStmt statement (continuation x {S}) is in Equation 7.15.*

$$\frac{F \Longrightarrow_f fn}{\textit{continuation } x\ \{S\} \Longrightarrow_s [t_2 = \textit{call create\_first\_continuation}(t_1), \textit{[CPS] call fn}(t_2)]} \tag{7.15}$$

In 7.15, F = void capture(Continuation x) {S} and t_1 is the continuation to the CPS call. For now t_1 is a place holder that will be replaced by the continuation to the CPS call instruction ([CPS] call fn(t_2)) in the CPS transformation.

Example 7.3.10. *The translation of CallStmt statement (call (E)) is in Equation 7.16:*

$$\frac{E \Longrightarrow_e < I, t >}{\textit{call } (E) \Longrightarrow_s (I, \textit{call call\_cont}(t))} \tag{7.16}$$

The CallStmt statement is translated into a call to call_cont function which calls the first class continuation t.

7.4 Code Generation of Functions

A function consists of a list of basic blocks and a frame. When entering the function, it creates a frame and copies the arguments from the Argument Passing Store to the frame. Then it jumps to the entry basic block to start the execution. The translation of the function body uses the Equations mentioned above.

7.5 The Factorial Continuation Example

The IR code of the *fact* function in Figure 6.7 is shown in Figure 7.5, 7.6.

In Figure 7.5, block 0 is the start point of the function. Line 3 reads the argument passing store pointer from the global variable @arg_pass_store into t_1. Line 4 reads the entry pointer number t_2 from t_1. Before the CPS transformation, a function has only one entry point (0^{th} entry pointer). If $t_2 == 0$, it jumps to block 1 and creates a frame on the AMIBE stack (The first argument to $create\_frame$ function is the size of the frame, it cannot be decided until the CPS transformation is completed and is temporarily set to 0). Line 18 loads the frame pointer from the argument passing store (The load seems redundant but it is necessary since after the CPS transformation, the frame pointer is passed as an argument on a continuation call. The function uses t_7 as the frame pointer without knowing it is entered by an ordinary function call or a continuation call).

Block 4 copies the arguments from the argument passing store to the frame and jumps to the entry basic block (block 5). Block 5 is translated from the conditional expression ($n == 0$) in the if statement. It checks the value of n. If it is 0 it jumps to block 6. Otherwise it jumps to block 9.

Block 6 contains the instructions translated from the continuation statement (continuation c $\{x[0] = c;\}$). Notice at line 41 the argument to $create\_first\_continuation$ is a place holder ($null$) which will be replaced by the continuation to the call at line 42 in the CPS transformation (discussed in Chapter 8). Block 6 falls through to block 7 which is translated from the return statement (return 1;).

In Figure 7.6, block 9 and 10 are translated from the statement (return $n * fact(n-1, x)$). In block 9, Line 3 is translated from n. Lines 4-7 are translated

from $fact(n-1, x)$. Block 9 falls through to block 10 which contains the code of the multiplication and return.

The translation of the *main* function is similar. The interesting part is the translation of the continuation call ($call(x[0])$;), as shown in Figure 7.7. Lines 2-4 load the first class continuation $x[0]$ into t_{17} (where t_7 is the frame pointer in *main*). Line 5 calls t_{17}.

```
1 define fact: (int, Continuation[])->int
2 Block 0 <From: ; To: 1 2 >:
3         t1 = ld @arg_pass_store
4         t2 = ld t1[0] // get the entry pointer number
5         t3 = t2==0
6         cbr t3 %frame_create% else %after_frm_create%
7
8 Block 1 <From: 0 ; To: 2 >:
9 frame_create:
10        t4 = ld t1[2]     //load continuation
11        t5 = ld t1[3]     //load access link
12        t6 = call create_frame(0,t4,t5)   //create the frame
13        st t1[1], t6      //store frame pointer to the argument passing store
14        br %after_frm_create%
15
16 Block 2 <From: 0 1 ; To: 4 3 >:
17 after_frm_create :
18        t7 = ld t1[1]     //load frame pointer
19        cbr t3 %initcall% else %sel_out%
20
21 Block 3 <From: 2 ; To: >:
22 sel_out :
23        halt('select  number out of range')
24
25 Block 4 <From: 2 ; To: 5 >:
26 initcall :
27        t8 = ld t1[4]     //copy arguments to frame
28        st t7[9], t8
29        t9 = ld t1[5]     //copy arguments to frame
30        st t7[10], t9
31        br %entry%
32
33 Block 5 <From: 4 ; To: 6 9 >:
34 entry:
35        t10 = ld t7[9]    //load variable n
36        t11 = t10==0
37        cbr t11 %iftrue% else %iffalse%
38
39 Block 6 <From: 5 ; To: 7 >:
40 iftrue :
41        t12 = call create_first_continuation(null) //create first class continuation
42        [CPS] call capture(t12) //capture the continuation
43
44 Block 7 <From: 6 ; To: >:
45 aftercall :
46        ret 1
47
48 Block 8 <From: ; To: 11 >:
49        br %endfunc%
```

Fig. 7.5: IR Code of the *fact* Function (Part 1)

```
1 Block 9 <From: 5 ; To: 10 >:
2  iffalse :
3          t13 = ld t7[9]    //load variable n
4          t14 = ld t7[9]    //load variable n
5          t15 = t14−1
6          t16 = ld t7[10]   //load variable x
7          [CPS] t20 = call fact(t15,t16)
8
9 Block 10 <From: 9 ; To: >:
10 aftercall1 :
11         t21 = t13∗t20
12         ret t21
13
14 Block 11 <From: 8 ; To: >:
15 endfunc:
16         halt('end of function')
```

Fig. 7.6: IR Code of the *fact* Function (Part 2)

```
1    ....
2          t15 = ld t7[6]    //load variable x
3          t16 = 0+1
4          t17 = ld t15[t16] //load array element
5          call  call_cont (t17)       //call to first class continuation
6    ....
```

Fig. 7.7: IR Code Segment of the *main* Function

Chapter 8

CPS Transformation

8.1 Introduction

The CPS transformation pass translates the IR code into CPS form by using the technology discussed in Chapter 4. The following sections discuss the CPS transformation of function calls and returns and the implementation of library functions for first class continuations.

8.2 CPS Transformation of Function Calls

In the IR code generation, a call to the function g in the function f is translated into the virtual instructions in Figure 8.1. Assume it is the i^{th} function call in f, then the block $k+1$ is the i^{th} continuation block.

The CPS transformation of the function call is shown as follows:

1. The return value t_j is transformed to a load of the return value from the argument passing store in block $k+1$, as shown at line 10 in Figure 8.2.

2. Create a continuation object $< f, i, t_7 >$ (where t_7 is the frame pointer) by calling *create_cont* function (shown at line 6 in Figure 8.3). If the call instruction is gener-

```
1  define f: ...
2      ....
3
4  Block k: ...
5      ....
6          t_j = [CPS] call g(t_{j+1}, ..., t_{j+m}}) // the i^th function call
7
8  Block k + 1 <From: k ; To: >: // the i^th continuation block
9   aftercall :
10     ....
```

Fig. 8.1: IR Code Segment of a Function Call

```
1  define f: ...
2      ....
3
4  Block k: ...
5      ....
6          [CPS] call g(t_{j+1}, ..., t_{j+m}}) // the i^th function call
7
8  Block k + 1 <From: k ; To: >: // the i^th continuation block
9   aftercall :
10         t_j = ld t_1[2]
11     ....
```

Fig. 8.2: Transform of the Return Value in a Function Call

```
1  define f: ...
2      ....
3
4  Block k: ...
5      ....
6          t_c = create_cont(f, i, t_7) //create the continuation
7          [CPS] call g(t_{j+1}, ..., t_{j+m}}) // the i^th function call
8
9  Block k + 1 <From: k ; To: >: // the i^th continuation block
10 aftercall :
11         t_j = ld t_1[2]
12     ....
```

Fig. 8.3: Create a Continuation

```
1  define f: ...
2      ....
3
4  Block k: ...
5      ....
6          t_c = create_cont(f, i, t_7) //create the continuation
7          t_d = call create_first_continuation(t_c) // create first class continuation
8          [CPS] call g(t_d) // the i^th function call
9
10 Block k + 1 <From: k ; To: >: // the i^th continuation block
11 aftercall :
12         t_j = ld t_1[2]
13     ....
```

Fig. 8.4: Replace the Argument to *create_first_continuation*

ated by a continuation statement, the null argument to *create_first_continuation* should be replaced by the continuation object (shown in Figure 8.4).

3. Store the entry point number 0, continuation, access link and actual arguments in the argument passing store. The call instruction becomes a plain call instruction with no arguments and no return value. Shown in Figure 8.5 (t_{ac} is the access link of the function g). As mentioned in Chapter 7, the callee will copy arguments from the argument passing store into its frame on an ordinary function call.

```
1  define f: ...
2     ....
3
4  Block k: ...
5     ....
6           t_c = create_cont(f, i, t_7) //create the continuation
7           st t_1[0], 0          //store the entry point number
8           st t_1[2], t_c //store the continuation
9           st t_1[3], t_ac //store the access link
10          st t_1[4], t_j+1 //store the actual argument
11           ....
12          st t_1[3 + m], t_j+m //store the actual argument
13          call g()                // the i^th function call
14          ret
15
16 Block k + 1 <From: k ; To: >: // the i^th continuation block
17  aftercall :
18          t_j = ld t_1[2]
19    ....
```

Fig. 8.5: Store Arguments in the Argument Passing Store

The redundant return instruction at line 14 is useful for the LLVM tail call optimization (discussed in Chapter 9).

4. Add a conditional branch block p that branches to the i^{th} continuation block if the entry point number argument is equal to i. And the predecessor of the i^{th} continuation block is set to block p. Shown in Figure 8.6.

5. Some temporary variables created before the function call might be used in the continuation block. A liveness analysis is used to find out these temporary variables. They are saved into the frame before the function call and loaded from the frame when used.

```
1 define f: ...
2    ....
3
4 Block p <From: .. ; To: k + 1 q >:
5 selentryi :
6          t_e = ld t_1[0]
7          t_{e+1} = t_e == i
8          cbr t_{e+1} %aftercall% else %sel_out%
9
10 Block q <From: p ; To: >:
11 sel_out :
12         halt('select  number out of range')
13   ....
14
15 Block k + 1 <From: p ; To: >: // the i^th continuation block
16 aftercall :
17         t_j = ld t_1[2]
18   ....
```

Fig. 8.6: Add a Conditional Branch Block for the i^{th} Entry Point

8.3 CPS Transformation of Returns

A return instruction (ret t) is transformed int a continuation call in Figure 8.7 (Again, t_1 is the pointer to the argument passing store and t_7 is the frame pointer). Line 1 loads the continuation from the frame into t_c. Lines 2-4 extract the function pointer, entry point number and the frame pointer from the continuation t_c. Lines 5-7 store the arguments of the continuation call to the argument passing store. Line 8 pops the current frame (and the continuation on the frame) from the AMIBE stack. Line 9 calls the function.

As we can see, the CPS transformation transforms function calls and returns into plain function calls. All the user defined functions have the type $() \rightarrow void$ after the CPS transformation. In the following discussions, we call a transformed ordinary function call a "function call" and a transformed return a "continuation call".

```
1    t_c = ld t_7[2]      //load continuation
2    t_f = ld t_c[0]      //get function pointer from continuation
3    t_i = ld t_c[1]      //get entry point number from continuation
4    t_s = ld t_c[2]      //get frame pointer from continuation
5    st t_1[0], t_i       //store the entry point number
6    st t_1[1], t_s       //store the frame pointer
7    st t_1[2], t         //store the return value
8    call delete_frame(t_7)    //delete frame
9    call t_f() //continuation call
10   ret
```

Fig. 8.7: CPS Transformation of the Return Instruction

8.4 The AMIBE Stack after the CPS Transformation

After the CPS transformation, a continuation is pushed on the AMIBE stack by the library function *create_cont* before a function call. And the function call pushes a frame on the AMIBE stack by the library function $create\_frame$. The frame and continuation is alive until a continuation call is reached. The call to the library function $delete\_frame$ pops the frame and the continuation from the AMIBE stack. Thus the AMIBE stack contains interleaving continuations and frames.

For instance, in function f the call to function g results in the changes of the AMIBE stack, as shown in Figure 8.8.

8.5 The Factorial Continuation Example

After the CPS transformation, the IR code of the function $fact$ in the factorial continuation program are in Figures 8.9, 8.10 and 8.11.

In Figure 8.9, block 3 and 4 are entry points for continuation calls. In Figure 8.10, block 8 contains the IR code of the continuation statement after the CPS transformation.

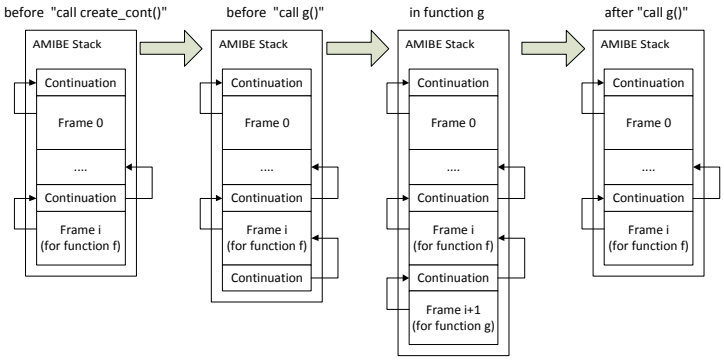


Fig. 8.8: State Changes of the AMIBE Stack

Notice the argument to *create_first_continuation* is replaced with the continuation to the call of *capture*. Block 9 contains the IR code of the (ret 1) after the CPS transformation.

In Figure 8.11, block 11 contains the IR code of the recursive call to $fact$ after the CPS transformation. Block 12 contains the IR code of another return instruction. The temporary variable returned by the load instruction at line 3 is used in the block 12. It is stored into the frame at line 4 and loaded back from the frame at line 23. Thus its value is correctly restored on a continuation call.

8.6 Implementation of Library Functions for First Class Continuations

After the code generation and the CPS transformation. A continuation statement is transformed into a function call that takes the return value of *create_first_continuation* as the argument. A continuation call is transformed into the a call to the *call_cont* library function. Their implementations are discussed in the following sections.

8.6.1 Implementation of *create_first_continuation*

The following call takes a continuation $c =< f, i, s >$ and returns a first class continuation object k that captures the AMIBE stack state.

```
1 k = call create_first_continuation(c)
```

The *create_first_continuation* does the following (shown in Figure 8.12):

1. Allocate a block of memory and copy the AMIBE stack to it.

2. Return the continuation c' on the copied stack (i.e., the first class continuation k is the continuation c' within a AMIBE stack snapshot).

8.6.2 Implementation of *call_cont*

The following call calls a first class contention $k =< f, i, s >$.

```
1 call call_cont(k)
```

It does the following (shown in Figure 8.13):

1. Find the head of the AMIBE stack snapshot in k. Copy it back to the AMIBE stack (excluding the last continuation).

2. Store the entry point number i and the frame pointer s in the argument passing store.

3. Call the function f

What it does is similar to the return instruction after the CPS transformation. Except that the AMIBE stack is restored before calling f.

The call to function f in the last step is followed by a redundant return instruction, as follows:

```
1  ...
2  call f()
3  ret
```

It is necessary for the tail call optimization in LLVM which will be discussed in Chapter 9.

```
1  define fact: ()->void
2  Block 0 <From: ; To: 1 2 >:
3          t1 = ld @arg_pass_store
4          t2 = ld t1[0] // get the entry pointer number
5          t3 = t2==0
6          cbr t3 %frame_create% else %after_frm_create%
7
8  Block 1 <From: 0 ; To: 2 >:
9  frame_create:
10         t4 = ld t1[2]     //load continuation
11         t5 = ld t1[3]     //load access link
12         t6 = call create_frame(12,t4,t5)  //create the frame
13         st t1[1], t6      //store frame pointer to the argument passing store
14         br %after_frm_create%
15
16 Block 2 <From: 0 1 ; To: 6 3 >:
17 after_frm_create :
18         t7 = ld t1[1]     //load frame pointer
19         cbr t3 %initcall% else %selentry%
20
21 Block 3 <From: 2 ; To: 9 4 >:
22 selentry :
23         t31 = ld t1[0]
24         t32 = t31==1
25         cbr t32 %aftercall% else %selentry1%
26
27 Block 4 <From: 3 ; To: 12 5 >:
28 selentry1 :
29         t34 = ld t1[0]
30         t35 = t34==2
31         cbr t35 %aftercall1% else %sel_out%
32
33 Block 5 <From: 4 ; To: >:
34 sel_out :
35         halt('select  number out of range')
36
37 Block 6 <From: 2 ; To: 7 >:
38  initcall :
39         t8 = ld t1[4]     //copy arguments to frame
40         st t7[9], t8
41         t9 = ld t1[5]     //copy arguments to frame
42         st t7[10], t9
43         br %entry%
44
45 Block 7 <From: 6 ; To: 8 11 >:
46 entry:
47         t10 = ld t7[9]    //load variable n
48         t11 = t10==0
49         cbr t11 %iftrue% else %iffalse%
```

Fig. 8.9: IR Code of the *fact* Function After the CPS Transformation (Part 1)

```
1  Block 8 <From: 7 ; To: 9 >:
2  iftrue :
3          t30 = call create_cont(fact,1,t7)  //create the continuation
4          t12 = call create_first_continuation(t30)  //create first class continuation
5          st t1[0], 0          //store the entry point number
6          st t1[2], t30        //store the continuation
7          st t1[3], t7         //store the access link
8          st t1[4], t12        //store actual arguments
9          call capture()  //capture the continuation
10         ret
11
12 Block 9 <From: 8 3 ; To: >:
13 aftercall :
14         t22 = ld t7[2]    //load continuation
15         t23 = ld t22[0]   //get function pointer from continuation
16         t24 = ld t22[1]   //get entry point number from continuation
17         t25 = ld t22[2]   //get frame pointer from continuation
18         st t1[0], t24        //store the entry point number
19         st t1[1], t25        //store the frame pointer
20         st t1[2], 1          //store the return value
21         call delete_frame(t7)    //delete frame
22         call t23() //continuation call
23         ret
24
25 Block 10 <From: ; To: 13 >:
26         br %endfunc%
```

Fig. 8.10: IR Code of the $fact$ Function After the CPS Transformation (Part 2)

```
1  Block 11 <From: 7 ; To: 12 >:
2   iffalse :
3           t36 = ld t7[9]     //load variable n
4           st t7[11], t36     //replaced temp write
5           t14 = ld t7[9]     //load variable n
6           t15 = t14−1
7           t16 = ld t7[10]    //load variable x
8           t17 = ld t7[3]     //follow access link
9           t18 = ld t17[11]   //load variable fact
10          t19 = ld t18[1]    //load access link
11          t33 = call create_cont(fact,2,t7)  //create the continuation
12          st t1[0], 0        //store the entry point number
13          st t1[2], t33      //store the continuation
14          st t1[3], t19      //store the access link
15          st t1[4], t15      //store the actual arguments
16          st t1[5], t16
17          call fact ()
18          return
19
20 Block 12 <From: 11 4 ; To: >:
21  aftercall1 :
22          t20 = ld t1[2]
23          t37 = ld t7[11]    //replaced temp read
24          t21 = t37*t20
25          t26 = ld t7[2]     //load continuation
26          t27 = ld t26[0]    //get function pointer from continuation
27          t28 = ld t26[1]    //get entry point number from continuation
28          t29 = ld t26[2]    //get frame pointer from continuation
29          st t1[0], t28      //store the entry point number
30          st t1[1], t29      //store the frame pointer
31          st t1[2], t21      //store the return value
32          call delete_frame(t7)     //delete frame
33          call t27() //continuation call
34          return
35
36 Block 13 <From: 10 ; To: >:
37 endfunc:
38          halt('end of function')
```

Fig. 8.11: The IR code for the $fact$ function After the CPS Transformation (Part 3)

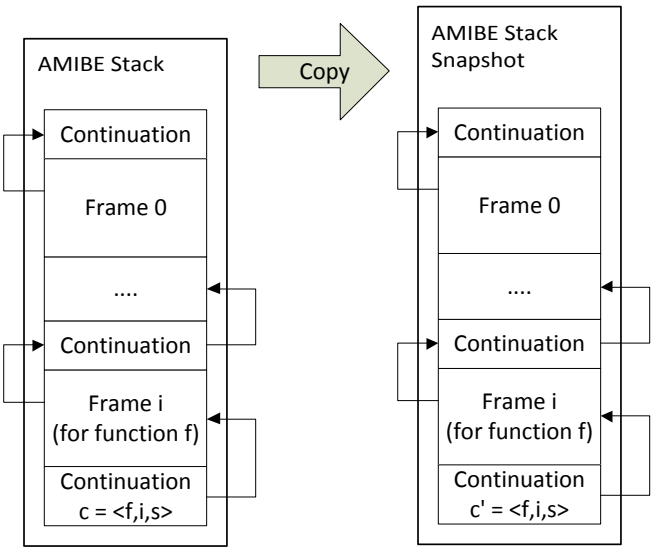


Fig. 8.12: Create a First Class Continuation

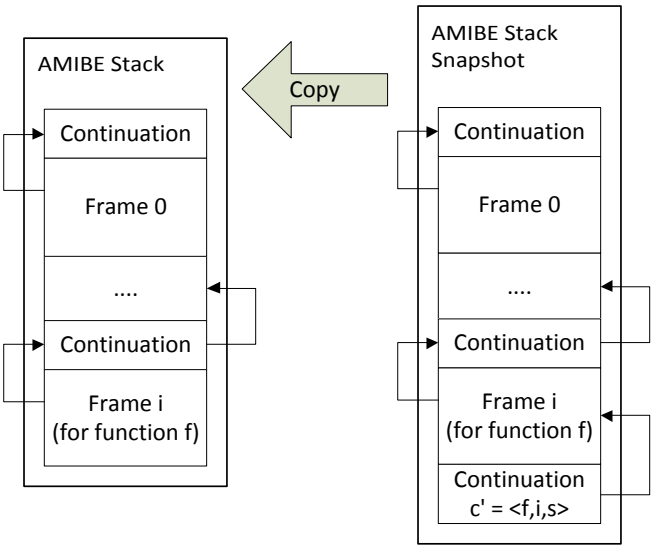


Fig. 8.13: Call a First Class Continuation

Chapter 9

LLVM Code Generation and Optimizations

9.1 Translation from the AMIBE IR to the LLVM IR

The AMIBE IR is designed to closely relate to the LLVM IR. Thus the translation of the AMIBE IR to LLVM IR is quite straightforward. The LLVM assembly language is introduced in [16] [17].

9.1.1 Translation of IR Values

As mentioned in Chapter 7, every AMIBE IR value occupies one cell with fixed length. In the LLVM IR every value is an integer. The size of a cell is the size of an integer value. In the following discussion, we assume the LLVM is running on a 64-bit machine. The type of a value in LLVM is i64.

The IR values are translated into LLVM values. Table 9.1 depicts a part of the translation. Notice that even real numbers and pointers are stored as integer. When used as operands, they are converted into their true types(real or pointer) on the fly. When returned by the computation, they are converted back to integers.

IR Value	LLVM Value
$Temp(id)$	a LLVM value id
$Mem(v_1, v_2)$	getelementptr i64* v_1, i64 v_2
$FuncVal(f)$	a LLVM function f
$ConstInt(i)$	i64 i
$ConstReal(r)$	i64 r
$Global(g)$	i64 g

Table 9.1: Part of the Translation of IR Values

9.1.2 Translation of IR Instructions

A part of the translation of IR instruction is shown in Table 9.2. Notice that after the CPS transformation, call and return instructions do not take any arguments.

Name	IR Instruction	LLVM Instruction
$Store$	st m, v	store i64 v, i64* m
$Load$	v = ld m	v = load i64* m
New	m = new i	m = call i64* @$imalloc$(i64 i)
$ICmp$	$v = v_1\ OP_c\ v_2$	v = icmp OP_c i64 v_1, v_2
$IAdd$	$v = v_1 + v_2$	v = add i64 v_1, v_2
$ISub$	$v = v_1 - v_2$	v = sub i64 v_1, v_2
$IMul$	$v = v_1 * v_2$	v = mul i64 v_1, v_2
$IDiv$	$v = v_1 / v_2$	v = div i64 v_1, v_2
$IMod$	$v = v_1 \% v_2$	v = srem i64 v_1, v_2
$Label$	l:	-
Br	br l	br label l
CBr	cbr v l_1 l_2	br i1 v, label l_1, label l_2
$Halt$	halt(msg)	call void @$print\_err\_msg$(i64 msg)
$Call$	call $f()$	call void $f()$
Ret	ret	ret void

Table 9.2: Part of the Translation of IR Instructions

9.2 Tail Call Optimization

In LLVM tail call optimization is possible when the following conditions are met [18]:

1. Caller and callee both have the calling convention fastcc.

2. The call is in tail position (ret immediately follows call and ret uses value of call or is void).

3. Option -tailcallopt is enabled, or llvm::GuaranteedTailCallOpt is true.

4. Platform specific constraints are met. On x86/x64 (where the AMIBE is tested on), no variable argument lists are used.

The AMIBE compiler meets all the 4 conditions by using the following techniques:

1. It marks the functions and function calls that are user-defined. In the LLVM code generation, it add the fastcc calling conventions to the functions and function calls.

2. After the CPS transformation, every user function call has a 'ret void' instruction following it, as mentioned in Chapter 7 and 8.

3. The -tailcallopt is enabled in the LLVM Static Compiler (llc) when generating executable codes.

4. As mentioned before, function calls do not take any arguments after the CPS transformation. Arguments are passed by using the argument passing store.

In the factorial continuation example, the LLVM code segment of the recursive call to $fact$ in the $fact$ function is shown as follows:

```
1   ...
2   call fastcc void @fact()
3   ret void
4
5 aftercall :                                   ; preds = %selentry
6   ...
```

The call instruction meets all the conditions for the tail call optimization. Then the LLVM code is compiled by the LLVM Static Compiler into assembly code with -tailcallopt enabled. On x86/x64, the function call is compiled into a jump, shown in the following code segment:

```
1   ...
2         jmp     fact                    // TAILCALL
3
4 .LBB18_12:                              // %aftercall
5   ...
```

After the LLVM code generation and tail call optimization, the call to user defined functions never make the system stack grow since they are all replaced by jumps (library function calls still push and pop data on the system stack). The technology of compiling with CPS in AMIBE is practical because the system stack would not grow indefinitely.

Chapter 10

Performance Evaluation

10.1 Introduction

This chapter compares the performance of the AMIBE compiler against a naive just-in-time compiler based on GNU Lightning and currently used by COMET. The tests are listed below:

1. The recursive factorial program
2. The recursive factorial program with first class continuations
3. The N-QUEEN program

These tests compare the time and space performance in AMIBE and COMET on a iMac with a 2.8GHz Intel Core 2 Duo processor and 2GB 667MHz DDR2 memories. The AMIBE program is compiled with Level 3 Optimization provided by the LLVM optimizing command line tool 'opt'.

10.2 Recursive Factorial Program

A recursive factorial program in AMIBE is given in Figure 10.1. It recursively computes the factorial of 10 n times. (All the test programs in COMET are exactly the

same as those in AMIBE with some syntactical differences). It is used to measure the performance of simple function calls in AMIBE.

```
1   int fact(int n)
2   {
3     if (n == 0) {
4       return 1;
5     }
6     else
7       return n*fact(n-1);
8   }
9
10  int main()
11  {
12    int n = 1000000;
13    for (int i = 0; i < n; i=i+1) {
14      fact(10);
15    }
16    return 0;
17  }
```

Fig. 10.1: Recursive Factorial Program in AMIBE

Table 10.1 depicts the time and space to compute the factorial of 10 in AMIBE and COMET with n increased from $1,000,000$ to $20,000,000$. For this program AMIBE is $1.77-2.19$ times as fast as the COMET. Both COMET and AMIBE use constant heap space. The default size of the AMIBE stack is 1MB. Thus the 1.06MB space used by AMIBE shows a very small space overhead.

	Language	$n = 10^6$	$n = 2 * 10^6$	$n = 5 * 10^6$	$n = 1 * 10^7$	$n = 2 * 10^7$
Time(s)	COMET	0.435	0.782	1.826	3.567	7.052
	AMIBE	0.198	0.402	0.983	1.984	3.976
Space	COMET	163.59KB	163.59KB	163.59KB	163.59KB	163.59KB
	AMIBE	1.06MB	1.06MB	1.06MB	1.06MB	1.06MB
Speedup		2.1970	1.9453	1.8576	1.7979	1.7736

Table 10.1: Recursive Factorial Program

10.3 Recursive Factorial Program with First Class Continuations

The second test computes the factorial of 10 n times with first class continuations, as shown in Figure 10.2 (it uses the library functions in Section 2.4.4). Once again, only the AMIBE program is shown. It is used to measure the performance of invoking first class continuations.

```
1   int fact(int n, Continuation[] x)
2   {
3     if (n == 0) {
4       continuation c {
5         x[0] = c;
6       }
7       return 1;
8     }
9     else
10      return n*fact(n-1, x);
11  }
12
13  int main()
14  {
15    int n = 1000000;
16    int[] iter = createHeapValue();
17
18    Continuation[] x = new Continuation[](1);
19    int d = fact(5, x);
20
21    incHeapValue(iter);
22    if (getHeapValue(iter) < n)
23      call(x[0]);
24
25    return 0;
26  }
```

Fig. 10.2: Recursive Factorial Program with First Class Continuations in AMIBE

Table 10.2 depicts the time and space to compute the factorial of 10 with first class continuation in AMIBE and Comet. AMIBE is $1.91 - 2.16$ times as fast as the Comet. Again both Comet and AMIBE use constant heap space.

	Language	$n = 10^6$	$n = 2 * 10^6$	$n = 5 * 10^6$	$n = 1 * 10^7$	$n = 2 * 10^7$
Time(s)	COMET	0.597	1.101	2.641	5.191	10.544
	AMIBE	0.276	0.544	1.358	2.709	5.423
Space	COMET	163.56KB	163.56KB	163.56KB	163.56KB	163.56KB
	AMIBE	1.06MB	1.06MB	1.06MB	1.06MB	1.06MB
Speedup		2.1630	2.0239	1.9448	1.9162	1.9443

Table 10.2: Recursive Factorial Program with Call to First Class Continuations

10.4 *N*-Queen Program

The test consists of 2 sub-tests. First the recursive *N*-QUEEN program is tested. It measures the performance of AMIBE vs. COMET on a plain DFS search. Second the simple backtracking *N*-QUEEN program using first class continuations is tested. It measures the performance of the AMIBE vs. COMET on a continuation based search. Finally the overhead of using first class continuations in COMET and AMIBE is calculated from the two tests.

10.4.1 Recursive *N*-Queen Program

The recursive *N*-QUEEN program in AMIBE is given in Figure 10.3 and 10.4 (COMET program is similar). It uses the forward checking algorithm [13] and depth-first-search strategy [9]. For the array q, $q[i]$ represents the column which the queen at row i is placed in. The function *next_placement* finds the next safe placement for $q[0], .., q[r]$ recursively. The *queen* function takes an array q and finds the first solution to the *N*-QUEEN problem. Then it finds all the other solutions until *next_placement* returns false.

The recursive *N*-QUEEN program has the same search strategy as the simple

backtracking N-Queen program. It is a base case for measuring the overhead of using firs class continuations. It also measures the performance of the DFS search in AMIBE against Comet.

Table 10.3 depicts the time and space of the recursive N-Queen program with n increased from 9 to 13. AMIBE is about 5 times as fast as Comet. Both Comet and AMIBE use constant heap space.

	Language	$n = 9$	$n = 10$	$n = 11$	$n = 12$	$n = 13$
Time(s)	Comet	0.125	0.286	1.206	6.858	43.292
	AMIBE	0.013	0.048	0.241	1.402	8.765
Space	Comet	164.20KB	164.20KB	164.20KB	164.14KB	164.05KB
	AMIBE	1.06MB	1.06MB	1.06MB	1.06MB	1.06MB
Speedup		9.6154	5.9583	5.0041	4.8916	4.9392

Table 10.3: Recursive N-Queen Program

10.4.2 Simple Backtracking N-Queen Program

The simple backtracking N-Queen program in AMIBE is given in Figure 2.8 and repeated here in 10.5 (Comet program is similar). It uses the library functions (such as *label* and *backtrack*) defined in Section 2.4.4. For the array q, $q[i]$ represents the column which the queen at row i is placed in. The *queen* function tries to place $q[i]$ one at a time. If there are conflicts, it backtracks with the last continuation and tries another decision. The search strategy is exactly the same as the recursive program. But there is the overhead to create and call first class continuations in the program.

Table 10.4 depicts the time and space of the simple backtracking N-Queen program with n increased from 9 to 13. AMIBE is about 4 times as fast as Comet. Both Comet and AMIBE use constant heap space.

```
1   // check if q[i] attacks q[j]
2   bool attack(int[] q, int i , int j)
3   {
4     return q[i]==q[j] || q[i]-q[j]==i-j || q[i]-q[j]==j-i;
5   }
6
7   // check if q[r] attacks q [0],.., q[r-1]
8   bool attackg(int[] q, int r)
9   {
10    for (int t = 0; t < r; t=t+1)
11      if (attack(q, t, r)) return true;
12    return false;
13  }
14
15  // find a safe column to place q[r] starting from q[r]+1 to n-1
16  bool find_placement(int[] q, int r)
17  {
18    int n = q.length;
19
20    for (int i = q[r]+1; i < n; i=i+1) {
21      // try to place q[r] in i
22      q[r] = i;
23      // check if q[r] attacks other queens
24      if (!attackg(q, r)) return true;
25    } // for
26
27    return false;
28  }
29
30  // find the next safe placement for q [0],.., q[r] recursively
31  bool next_placement(int[] q, int r)
32  {
33    if (r < 0) return false;
34
35    if (find_placement(q, r)) return true;
36    q[r] = -1; // reset q[r]
37
38    while (next_placement(q, r-1)) {
39      if (find_placement(q, r)) return true;
40      q[r] = -1; // reset q[r]
41    }
42
43    return false;
44  }
```

Fig. 10.3: Library Functions for the Recursive N-Queen Program in AMIBE

```
1   // nqueen function
2   int queen(int[] q)
3   {
4     int n = q.length;
5     int nSols = 0;
6
7     // find the first solution
8     bool reachN = true;
9     for (int i = 0; i < n; i=i+1) {
10      if (!next_placement(q, i)) {
11        reachN = false;
12        break;
13      }
14    }
15
16    if (!reachN) return nSols;
17    nSols=nSols+1;
18
19    //find all the following solutions
20    while (next_placement(q, n−1))
21      nSols=nSols+1;
22    return nSols;
23  }
24
25  int main()
26  {
27    int n = 11;
28    int[] q = new int[](n);
29    // initialize the placement of queens
30    for (int i = 0; i < n; i=i+1) q[i] = −1;
31
32    printi(queen(q)); // print the number of solutions
33    return 0;
34  }
```

Fig. 10.4: Recursive N-Queen Program in AMIBE

	Language	$n = 9$	$n = 10$	$n = 11$	$n = 12$	$n = 13$
Time(s)	Comet	0.197	0.605	2.808	15.570	94.928
	AMIBE	0.033	0.141	0.723	4.060	24.512
Space	Comet	164.89KB	396.83KB	396.77KB	396.83KB	396.77KB
	AMIBE	1.06MB	1.06MB	1.06MB	1.06MB	1.06MB
Speedup		5.9697	4.2908	3.8838	3.8350	3.8727

Table 10.4: Simple Backtracking N-Queen Program

```
1   // queen i attacks queen j
2   bool attack(int[] q, int i, int j)
3   {
4     return q[i]==q[j] || q[i]-q[j]==i-j || q[i]-q[j]==j-i;
5   }
6
7   void queen(int[] q, Continuation[] contStack, int[] stackSize)
8   {
9     int n = q.length;
10    for (int i = 0; i < n; i=i+1) {
11      q[i] = label(0, n-1, contStack, stackSize);
12      for (int j=0; j < i; j=j+1) {
13        if (attack(q, i, j))
14        backtrack(contStack, stackSize);
15      }
16    } // for
17  }
18
19  int main()
20  {
21    Continuation[] contStack = new Continuation[](100);
22    int[] stackSize = createHeapValue();
23
24    int n = 9;
25    int[] nSols = createHeapValue();
26
27    bool exit = true;
28    continuation exitPoint {
29      pushStack(contStack, stackSize, exitPoint);
30      exit = false;
31    }
32
33    if (!exit) {
34      int[] q = new int[](n);
35      queen(q, contStack, stackSize);
36      incHeapValue(nSols);
37      backtrack(contStack, stackSize);
38    }
39    else {
40      printi(getHeapValue(nSols));
41      return 0;
42    }
43  }
```

Fig. 10.5: Simple Backtracking N-Queen Program in AMIBE

10.4.3 Overhead of First Class Continuations

The overhead of using first class continuations is computed by the following equation:

$$Overhead = \frac{time\_of\_backtracking\_program - time\_of\_recursive\_program}{time\_of\_recursive\_program} \quad (10.1)$$

Table 10.5 depicts the overhead of the N-Queen program in AMIBE and Comet computed from the two tests above. The overhead in AMIBE is significantly larger than the one in Comet. That is because AMIBE uses a naive strategy of saving and restoring the entire stack for continuations, which wastes a large amount of CPU time in the memory allocation.

Even though the overhead of using first class continuations is big, the performance of AMIBE is 4 times as fast as Comet (shown in Table 10.4), which is attributed to the extensive optimizations provided by LLVM.

	Language	$n = 9$	$n = 10$	$n = 11$	$n = 12$	$n = 13$
Overhead	Comet	0.5760	1.1154	1.3284	1.2703	1.1927
	AMIBE	1.5385	1.9375	2.0000	1.8959	1.7966

Table 10.5: Overhead of Using First Class Continuations in the N-Queen Program

10.5 Summary

As the sections above show, AMIBE outperforms Comet in all test cases, with or without first class continuations. The performance increase is bigger as the program becomes more complex, which is the benefit of the extensive optimizations provided by LLVM. Because of the naive strategy of saving and restoring stacks in AMIBE, the overhead of using first class continuations in AMIBE is significantly larger than in Comet. Nonetheless, the performance of programs using first class continuations in AMIBE is still much better than in Comet. With a better implementation strategy for first class continuations, further improvements can be achieved in AMIBE.

Chapter 11

Conclusion

This thesis developed an imperative programming language AMIBE with first class continuations. AMIBE uses a modern compiler infrastructure LLVM that provides a collection of optimizing tools and technologies. To implement first class continuations using LLVM which does not allow explicit manipulations of the system stack, AMIBE is compiled with Continuation Passing Style (CPS) to avoid copying the system stack. In this thesis, the technologies for compiling with CPS in AMIBE is developed and the implementation of first class continuations is discussed. The performance of AMIBE is compared against COMET that uses GNU Lightning. Test results show that AMIBE outperforms COMET several times in CPU Time.

The technology of compiling with CPS to avoid copying the system stack can be used to implement first class continuations in any compiler infrastructures that provide the tail call optimization.

11.1 Future Work

Currently, AMIBE saves and restores the entire stack in first class continuations. The performance of the memory accesses can be improved by saving and restoring the stack

incrementally.

Also, it is not necessary to compile with CPS for functions that do not contain first class continuations or call other functions that contain first class continuations indirectly. That is, compiling with CPS could be done on an as-needed basis.

The procedural programming structures in AMIBE has been developed. In the future the structures for object-oriented programming will be developed. OO is convenient for the development of libraries for non-deterministic searches.

Bibliography

[1] Bison - GNU parser generator, 2008. `http://www.gnu.org/software/bison/`.

[2] flex: The Fast Lexical Analyzer, 2008. `http://flex.sourceforge.net/`.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

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

[5] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java, 2nd edition*. Cambridge University Press, 2002.

[6] Paolo Bonzini. GNU lightning, 2004. `http://www.gnu.org/software/lightning/`.

[7] W Clinger, D P Friedman, and M Wand. *A scheme for a higher-level semantic algebra*, pages 237–250. Cambridge University Press, New York, NY, USA, 1986.

[8] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher Order Symbol. Comput.*, 12:7–45, April 1999.

[9] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[10] Olivier Danvy and Lasse R. Nielsen. A first-order one-pass cps transformation. *Theor. Comput. Sci.*, 308:239–257, November 2003.

[11] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. *SIGPLAN Not.*, 39:502–514, April 2004.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *Proceedings of the 6th international joint conference on Artificial intelligence - Volume 1*, pages 356–364, San Francisco, CA, USA, 1979. Morgan Kaufmann Publishers Inc.

[14] Pascal Van Hentenryck and Laurent Michel. Nondeterministic control for hybrid search. *Constraints*, 11:353–373, December 2006.

[15] Andrew Kennedy. Compiling with continuations, continued. *SIGPLAN Not.*, 42:177–190, October 2007.

[16] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[17] Chris Lattner. LLVM Assembly Language Reference Manual, 2011. `http://llvm.org/docs/LangRef.html`.

[18] Chris Lattner. The LLVM Target-Independent Code Generator, 2011. `http://llvm.org/docs/CodeGenerator.html#tailcallopt`.

[19] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[20] L. Michel, A. See, and P. Van Hentenryck. High-level nondeterministic abstractions in c++. In *12th International Conference on Principles and Practice of Constraint Programming. (CP'06)*, Lecture Notes in Computer Science, Nantes, France, September 2006.

[21] John C. Reynolds, A. Van Wijngaarden, A. W. Mazurkiewicz, F. L. Morris, C. P. Wadsworth, J. H. Morris, M. J. Fischer, and S. K. Abdali. The discoveries of continuations, 1993.

[22] S. Arun-Kumar Sanjiva Prasad. An introduction to operational semantics, 2003.

[23] Olin Shivers and Matthew Might. Continuations and transducer composition. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 295–307, New York, NY, USA, 2006. ACM.

[24] Gerald Jay Sussman. Scheme: An interpreter for extended lambda calculus. In *Memo 349, MIT AI Lab*, 1975.

[25] Pascal Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999.

[26] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.