Extending the UHC LLVM backend:
Adding support for accurate garbage collection

Paul van der Ende

MSc Thesis

October 18, 2010

INF/SCR-09-59

Universiteit Utrecht

Center for Software Technology
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

Daily Supervisors:
dr. A. Dijkstra
drs. J.D. Fokker

Second Supervisor:
prof. dr. S.D. Swierstra

Abstract

The Utrecht Haskell Compiler has an LLVM backend for whole program analyse mode. This backend previously used the Boehm-Weiser library for heap allocation and conservative garbage collection. Recently an accurate garbage collector for the UHC compiler has been developed. We wanted this new garbage collection library to work with the LLVM backend. But since this new collector is accurate, it needs cooperation with the LLVM backend. Functionality needs to be added to find the set of root references and traversing all live references.

To find the root set, the bytecode interpreter backend currently uses static stack maps. This is where the problem arises. The LLVM compiler is known to do various aggressive transformations. These optimizations might change the stack layout described by the static stack map. So to avoid this problem we wanted to use the shadow-stack provided by the LLVM framework. This is a dynamic structure maintained on the stack to safely find the garbage collection roots.

We expect that the shadow-stack approach comes with a runtime overhead for maintaining this information dynamically on the stack. So we did measure the impact of this method. We also measured the performance improvement of the new garbage collection method used and compare it with the other UHC backends.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Compilers are a success story of computer science. Compilers are typically software programs written by and for computer experts. This is one of the reasons this subject is well understood. This understanding has lead to well concerned architectures of compiler systems. Most compiler architectures are defined as a pipeline. The compilation process is defined as a sequence of steps, transforming the source language to the target language step by step. An example of a simple compiler pipeline could be: first parsing, then typechecking, desugaring, optimizations and finally code generation. Also variations to a pipeline could exist. If for generating machine code for different hardware platforms should be possible, the final step of our example pipeline could be split into multiple different code generation backends. Other variations to the compiler pipeline could exist for functional, historical or experimental reasons.

1.1 UHC

The Utrecht Haskell Compiler is a compiler for Haskell, developed at Utrecht University [DFS09]. It supports most Haskell 98 features, and some language extensions. Haskell is a polymorphically typed, non-strict, purely functional programming language. UHC is an alternative for the well known Glasgow Haskell Compiler (GHC). GHC is a production quality compiler, but UHC has a different focus. UHC has a systematical design and the architecture is easy to extend. So this allows one to experiment with new features. Experimentation is the main reason why the UHC has different modes of operation.

1.1.1 Intermediate languages

Intermediate languages that play an important role during translation from Haskell to the target language are Core, Grin and Silly. Core is a non-typed desugared simplification of Haskell. Grin is a small functional language with strict semantics. Silly is a simple imperative language. Transformations between these languages prepare Haskell for generating code that can be run on a hardware architecture, since these architectures are strict an imperative in nature.

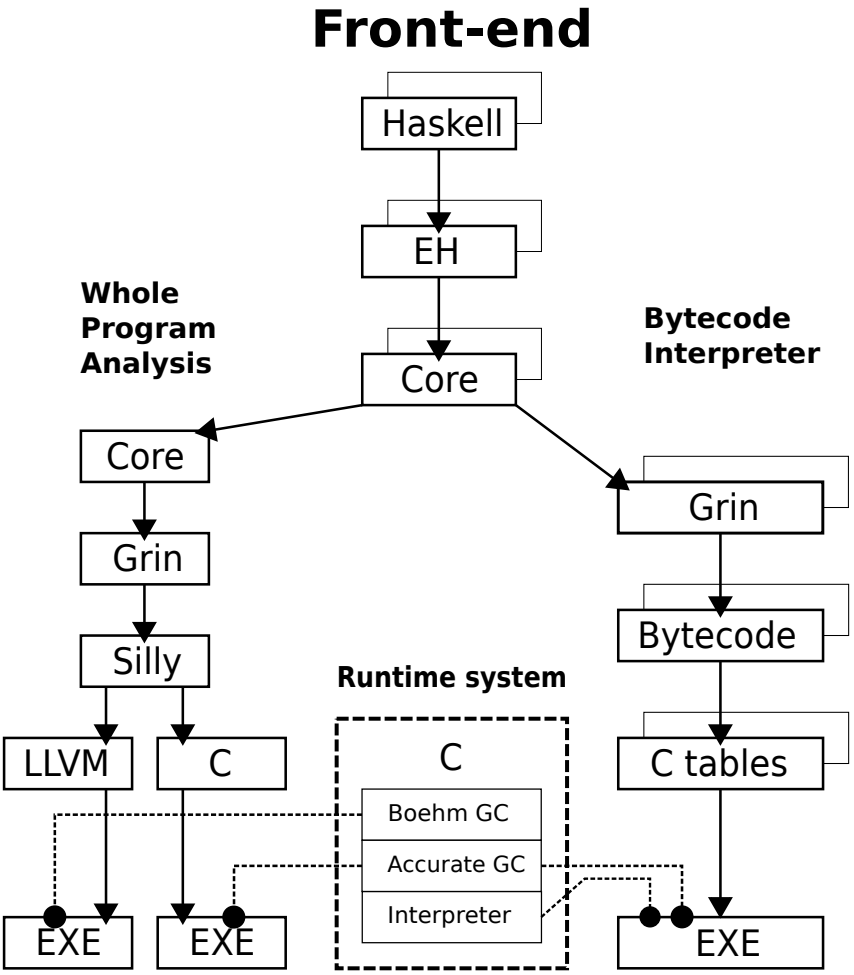


Figure 1.1: UHC pipeline

1.1.2 Modes of operation

The Utrecht Haskell Compiler pipeline can be split into three modes of operation [DFS09]. All modes share the same front end. The front end consists of a parser. But after the translation to Core, there is a choice from three modes of operation. Two modes of operation are relevant for this research project:

Whole-program analysis mode In this mode all intermediate code from the program modules and needed libraries are assembled together and processed further as a whole. This mode benefits from some optimizations that are possible only when all source code is available for analyse. This backend targets multiple target languages for experimentation. Targets currently available are C code, Common Intermediate Language (CIL) instructions [ECM06] and Low-Level virtual Machine (LLVM) instructions. The latter is important for this research project. See section 1.3 for more information about the LLVM framework. Compiling to other low level languages gives us the advantage that we do not have to deal with the low level aspects of different machine architectures. Optimising programs for different hardware architectures is not the main focus of the UHC project. We therefore benefit from for example the LLVM project.

Bytecode interpreter mode In this mode the source code modules are translated separately. We can not benefit from whole-program analyse, but there is another practical advantage. Suppose you want to compile a large multi module application. When you want to make changes to a single module, you do not need to re-analyse the whole program. You do not want this because it is a very time consuming process. This mode is more suitable for compiling applications during development. This backend generates C code with arrays of custom byte code. The byte code is interpreted at runtime by an interpreter. The generated C code is compiled to a native binary. A runtime system is linked with the binaries for low level functionality. The runtime also performs garbage collection for automatic memory management. This mode can be seen as the default mode.

1.2 Automatic memory management

In the early days programmers dealt manually with memory allocation and deallocation. This was often error prone and complicated the program design. An example of a design problem is that it can be hard to decide when a particular object in memory can be deallocated when having shared resources. Who is responsible for deciding the object is not needed any more so it can be deallocated? This question is hard to answer at the time of writing your program.
This leads to the introduction of automatic memory management. The programmer decides how much memory he needs, and the application itself decides when it is not needed anymore so the memory can be reclaimed. The process of finding obsolete memory is called garbage collection.

1.2.1 Advantages

Automatic memory management offers some advantages we like to discuss.

Abstraction Automatic memory management results in separation of concerns. The programmer does not have to think about memory deallocation so this simplifies the software design. Letting the runtime deal with memory management is a nice abstraction.

Robustness Automatic memory management prevents certain bugs. These bugs can cause software to crash, but memory usage bugs are also often abused by hackers. Automatic memory management will make your programs more secure. Well known memory allocation bugs are memory leaks. This happens when no more pointers to a piece of memory exist, so it can never be deallocated. The piece of memory is lost and can not be reclaimed. Another example are dangling pointers. A dangling pointer points to a piece of memory that is already deallocated. Reading and writing to the location of the pointer causes unpredictable behaviour. When correctly implemented, automatic memory management can prevent all these problems.

Performance Most conventional memory allocation algorithms like the implementations of the standard C malloc function causes the memory to be fragmented over time. Fragmentation happens when smaller fragments of allocated memory are scattered around in memory, so if a larger piece of memory is required, there is no suitable fit possible. This can be solved by storing larger objects in fragments, but your program ends up with a large administration of small fragments and this causes a performance penalty. Memory management systems that use a copying approach will prevent fragmentation. With this approach allocated memory is moved or copied very dense together during a garbage collection cycle. This results in better performance because new memory allocation are not fragmented so new memory could be allocated more effective.

There is another advantage of automatic memory management in the context of functional programming. Since UHC is a compiler for a functional language this advantage plays a important role. In a functional language it is often not known when a resource is not needed any more. The prevalence of sharing and delayed execution of suspensions means that functional languages often have particularly unpredictable execution orders [JL07]. So for functional programming it is often necessary to use automatic memory management.

1.2.2 Disadvantages

Automatic memory management also has some disadvantages. At certain points during the run of the program, the garbage collection algorithm has to perform garbage collection, or the program will eventually run out of memory. It is often unspecified or unpredictable when garbage collection will occur at runtime. Also the garbage collection cycle can take a long time to complete a memory scan when large amounts of memory are allocated. Garbage collectors that take a "stop-the-world" approach will stop the main application thread completely and this is sometimes not desirable. Even when using incremental of concurrent garbage collectors, overall performance during garbage collection will be reduced.

Besides the unpredictable behavior there is another problem. The time needed for performing a garbage collection cycle is divided in two parts. First the memory is analyzed, then the garbage is freed. The first analyze phase is overhead because this phase is not needed for custom memory management. So garbage collection comes already with an overhead compared to manual memory management.

Another problem with garbage collection is cache misses. Because the garbage collector moves memory around, and also touches a lot of memory when performing a memory scan, garbage collected programs have a poor locality. Programs can benefit from locality by caching often used memory to the processors cache. The processor cache can be accessed a lot faster so good locality means better performance in practice. Garbage collection influences the locality negatively, so the chance that a particular needed value is available in the cache is small.

1.3 LLVM

LLVM (Low Level Virtual Machine) is a compiler framework [LA04]. The goal of LLVM is to provide a framework for generating efficient machine code. The LLVM frameworks contains a compiler. This compiler focuses on low level optimizations, in contrast to the UHC compiler. The UHC compiler want to provide a framework for experimenting with language implementations and high level language features, and does not want to focus on machine specific details. The UHC compiler wants to use the LLVM framework to create efficient native executables for various platforms. The LLVM framework specifies a virtual machine and an instruction set for this machine. The instructions are used as input for the LLVM compiler. We explain more about the LLVM framework in chapter 3.

1.4 Problem description and motivation

We first describe the starting point of this project. The UHC project contains a compiler with various backends and modes of operation that are already working. A large set of Haskell programs can already be compiled and this will produce working executables. We described two modes of operation in section 1.1.2. We will now describe the technical details of relevant modes of operation and backend, with a special focus on memory management.

We explained in section 1.2.1 the advantages of automatic memory management and we want to benefit from this by having support for garbage collection for all backends. A garbage collector implementation is separate component and could be used by multiple backends. Currently UHC projects contains two libraries: one library for conventional garbage collection and one library for accurate garbage collection. For conventional garbage collection we use the external Boehm garbage collection library. The differences between accurate and conventional garbage collection are explained in the next chapter. The accurate garbage collection library is developed as part of the UHC project. Both libraries are part of the runtime system that is available for use by all compiled programs.

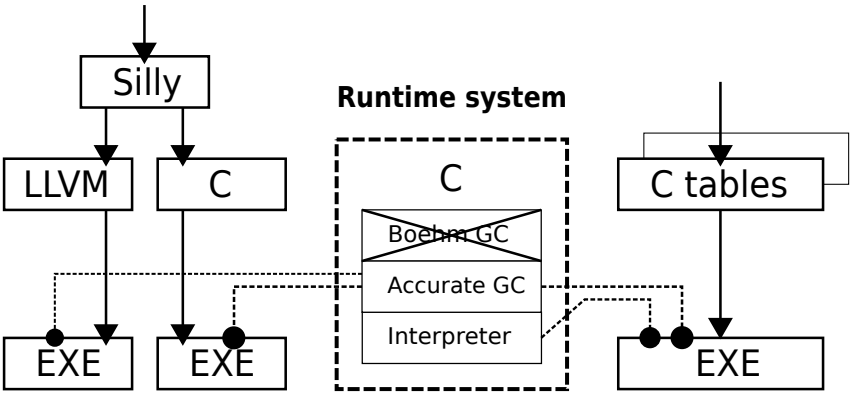


Figure 1.2: New UHC pipeline

Currently the bytecode interpreter mode of operation uses the accurate garbage collection library. Also the C backend of the whole program analys mode uses this accurate garbage collection library. The LLVM backend of the whole program analyse mode still uses the conventional Boehm garbage collection. We also want the LLVM backend to use the accurate garbage collection library. To be able to make this possible we need to investigate some important aspects:

- What is the impact on the LLVM backend? We need to know whether we can change the LLVM backend so it can cooperate with the accurate garbage collection library. We try to find the answer for this question in chapter 2.
- What role can the LLVM framework play in this project? We know the LLVM framework has support for garbage collection. We need to know how we can exploit this. We try to find the answer for this question in chapter 3.
- An additional aspect that plays a role for this research project is caused by the nature of the LLVM framework. The LLVM compiler is known to do many aggressive optimizations. We want to know how we could implement accurate garbage collection in such a way it is safe for the optimizations the LLVM compiler performs. We try to find the answer for this question in chapter 3.
- What is the impact on the accurate garbage collection library? We need to know if we need to modify or extend this library for cooperation with the LLVM backend. We try to find the answer for this question in chapter 4.

There are multiple reasons why we want to replace the conventional library with the accurate library for the LLVM backend.

- We expect the accurate garbage collection library to have better performance. Reasons for this are explained in section 2.2.4.
- We expect less failures caused by the conservative library. The accurate approach is proven to be correct.
- From engineering perspective there are also advantages. If both backends use the same garbage collector, they share the same code. Code sharing has big advantages. You have a smaller code base and no redundancy. Both backends benefit from new features, optimizations and bug fixes to the library. There is no need to maintain two separate libraries.

Chapter 2

Garbage collection

For automatic memory management we at some point need to know what memory is not used any more. This unused memory can be reclaimed and reused by the program itself or by other programs. This process of automatically freeing unused memory is called garbage collection. Many approaches to garbage collection exist. All approaches have their specific properties. We will first look into the basic principles of memory allocation and garbage collection. Our focus will be on theory needed for understanding how to create a compiler backend that can cooperate with a garbage collector. The garbage collector is part of the runtime. With cooperation we mean that the compiler backend will emit code that cooperates with the garbage collector at runtime. We will explain the basics of garbage collection and discuss some algorithms.

2.1 Memory allocation

Most computer architectures have different levels of memory. Examples are the main memory, cache and registers. From a the perspective of a programmer, the main memory is most important. Modern programming languages use a combination of stack and heap memory allocation model. Combining these models offers us more flexible memory allocation. We will first explain how the different memory models work. In early days only static allocation was used. This means that fixed amounts of memory where statically declared at compile that could not be changed at runtime. This is easy to implement and often very fast, but not very flexible. The largest disadvantage is that the size of data structures must be know at compile time and could not be dynamically created. The heap and the stack are often managed by the operating system and are located in the main memory. This model can been seen in figure 2.1. We can see the program code loaded into the memory, accompanied with static declarations. The amount of memory needed for this is fixed for an application. The heap and the stack are used for dynamic declarations. They can grow when the executable runs. Often they used in a way that they will grow towards each other to optimally divide the memory in two parts.

Besides main memory data structures like the stack and the heap, there are more levels of memory available. CPU's often have a cache memory. This is fast accessible memory that is used for data that is often needed. A computer programmer of high level languages often can influence where objects will be placed in memory. But he has no direct control of the cache. The cache is completely under control of the processor itself. The processor tries to predict what memory will be needed and stores this in the cache. The processor can be helped by presenting the data in such a way it is likely to benefit from the cache. Cache behavior is something we have to take into account when implementing a garbage collector. The CPU also has a very small piece of special purpose memory called registers. Registers play a larger role for garbage collection. This will be explained later in this section.

We will now describe the important data structures in more detail and explain what role they play for automatic memory management.

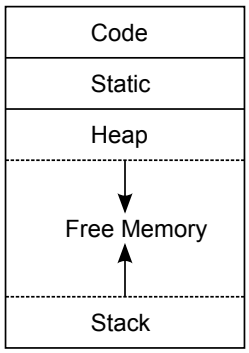


Figure 2.1: Typical memory layout

Stack allocation The stack is a data structure used for basic memory allocation. It is used by primitive operations to store intermediate values, local variables and storing administration for calling functions and returning results. Some CPUs are designed for making use of a stack. These CPUs have special registers for stack management, and most CPU instructions directly access the stack for reading and writing data. Most modern operating systems reserve a special piece of memory for the stack. The stack is a last-in-first-out (LIFO) data structure. The stack is especially suitable for temporary data which is not longer needed when the programs moves out of lexical scope. The code that uses the stack must clear the memory manually. A compiler will generate code that automatically frees stack space when it has been used. Memory on the stack is only used for local values and will not for values that are shared by the whole program. The stack thus needs no special memory management like garbage collection. But the stack plays an important role for garbage collection. This concept is explained further in section 2.2.1.

Heap allocation The heap is large pool of free memory. Heap memory provides more flexibility that the stack because it has no LIFO structure. Dynamically sized pieces of memory can be allocated with a dynamic lifetime. Some advantages of heap allocation are:

- Objects can be represented as hierarchical and recursive data structures.
- The size of memory objects is not fixed because they can be dynamically fitted.
- Procedures can return dynamically sized objects that need to be shared.
- Memory for special language constructs like closures and partially applied functions, often used by functional languages can be used.

Memory on the heap needs to be explicitly reclaimed. Because of the dynamic lifetime it is often not known when memory is not used any more. The heap is therefore very suitable for garbage collection.

Register allocation CPUs can have a limited amount of own storage that is located on the CPU itself. This piece of memory is called the registers and can be accessed very fast. Smart usage of registers can improve execution performance and is of interest for many low-level compiler optimizations. Some registers have special purposes, for example the stack pointer which marks the top of the stack. But some CPUs have general purpose registers that can be used by calling conventions to pass arguments to functions. Because the amount of registers is very limited, management of register is more related to compiler optimizations instead of garbage collection. The compiler will make sure the registers are efficiently used. But from garbage collection perspective registers play an large role. The compiler can decide to move memory between registers and the main memory. This can cause unpredictable behavior. A garbage collector can suffer from this and might not work correctly. So it is import to take this in mind.

2.2 Tracing garbage collection

As mentioned earlier in the section about memory allocation 2.1, we want to use garbage collection to manage the heap. There are different approaches to garbage collection. The most important approach is tracing garbage collection. Tracing garbage collection use a reachability to decide what memory has become garbage. This concept is very important for this thesis and is explained in next section. Another approach to garbage collection is for example reference counting.

2.2.1 Reachability

Memory that is not free but neither is needed any more is called garbage. But how do we know which memory has become garbage? Instead of finding out what memory is garbage, we could also look for memory that is not garbage. This is often easier to tell. If we know which data is live, we can simply keep this memory and dispose the rest. But how do we find memory that is still in use? We first define live memory as memory that is still in use. With memory that is still in use we mean memory that is still reachable. Live objects in memory will typically point to each other. Objects pointed to by other live objects are also live. Memory objects pointing to each other are forming a graph like structure. We can simply traverse the graph of live pointers and find all memory that is still reachable. Before this can be done, we need to know where to start looking. A set of memory locations is needed where to start the search for live objects.

Locations to start our search for live pointers can be found for example at registers, global declarations and the stack. This initial set of pointers is defined as the root set. One difficulty of implementing correct garbage collection is finding the root set at runtime, especially when the compiler does optimizations. Finding the root set can be done in various ways. A possible way to do it is by using conventions for memory usage, or by letting the compiler generate descriptors where to find the root set. Note that memory that is not reachable is not safe for garbage collection. It could be considered garbage and will be deleted during garbage collection.

Besides finding the root set, the garbage collector also must be able to trace the memory. This means it must be able to find a new set of pointers inside the root set to follow next. The garbage collector needs to understand the layout of the in memory objects, to find the new set of pointers.

2.2.2 Components

A program with automatic memory management using garbage collection consists of a few components working together. We first list all components and then we explain some more detail.

Heap Shared piece of memory used by the mutator for allocations.

Mutator The program using the garbage collector is called the mutator.

Garbage collector The garbage collector consists of three subcomponents.

- **Allocator** The allocator is responsible for allocating memory in the heap.
- **Tracer** The tracer is responsible for identifying all live memory.
- **Collector** The collector performs the garbage collection cycle to disposes all garbage.

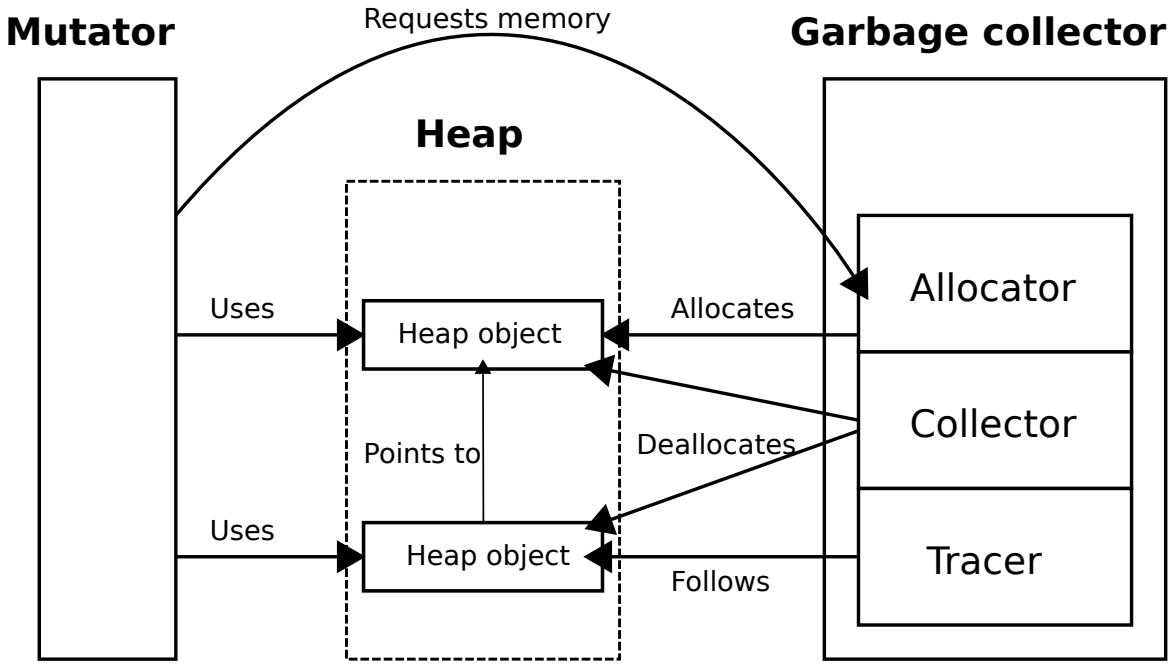


Figure 2.2: Automatic memory management components

2.2.3 Allocator

A garbage collector must implement its own heap memory allocator. This is needed because the garbage collector must have full control of all heap memory. It needs full control because it must be able to identify all allocated memory. The allocator usually abstracts over the way memory is allocated. The mutator only asks the allocator for a specific amount of memory, and the allocator must reserve enough space and return a reference to it. A very simple allocator could be system malloc. But this is generally not a good idea because this will result in large contiguous pieces of memory. In practice this will cause troubles because memory suffers from fragmentation and large pieces of memory are not likely to fit. So the allocator abstract over memory by providing a simple interface. But the implementation is build on top of pages, fragments and spaces. The allocator is responsible for initializing the garbage collector algorithm when the system is running low on free memory.

2.2.4 Tracer

The tracer must identify all references to live memory. These references could occur in different places in memory, as explained in section 2.2.1. The challenge for the tracer is to give meaning to the values in memory, and decide whether it is an reference or not. We distinguish two approaches for this:

Conservative A conservative tracer recognizes pointers by looking for particular values/bit patterns. When only looking to the actual value in memory, it is hard to fully guarantee that a given value is a pointer. Additional checks are done, for example check if the pointer really points to the heap. But some integer value could easily mistaken because it also qualifies these conditions. The Boehm garbage collector currently used in the LLVM backend of the UHC takes this approach.

Accurate Accurate tracers use some sort of exact technique to distinguish pointers from other values. This usually means that the mutator must do something extra to make this possible. The mutator must pass the garbage collector extra information that describes the values that are written to the heap. This information must make clear what values are pointers. The pointer information is used to be able to always trace the memory correctly.

Conservative versus Accurate

An advantage of conservative tracing is that it is straightforward to implement. There is no cooperation needed between the mutator and the garbage collector. Unfortunately there are also some disadvantages with the conservative tracing. Some values on the heap might be mistakenly recognized as a pointer. To be absolutely sure about a value being a pointer would require cooperation with the program allocating the memory, for example some kind of tagging method can be used for this, as used by accurate tracing. But the conservative approach does not allows you to do this because it needs cooperation with the mutator. Some faulty recognized pointers will be followed and this may cause unnecessary memory consumption. But this behavior is preferred above marking non-garbage as garbage because this would introduce unpredictable behavior and crashes. Also qualifying too much pointers will increase the search space. The heap scan will take longer because it has to follow all incorrectly found pointers too. This will increase the time of performing the garbage collection.

Another problem is correctness. A conservative scanner tries to recognize pointers. Although it takes a pessimistic approach of preferring marking pointers above non-pointers, it could still occur that a pointer is mistakenly not recognized as a pointer. This results in a reachable value becoming garbage, and this can cause the mutator to crash. Using conservative tracing can not guarantee that is always works, and this is not desirable.

But accurate tracing also has a disadvantage. Cooperation with the compiler backend is required. This makes the design of the compiler more complex. It also puts a runtime overhead on the mutator because it has to do extra things to make garbage collection possible.

2.2.5 Collector

The responsibility of the collector is to free all garbage. It will use the tracer to find all live references. The live memory objects need to be preserved, and all garbage memory objects need to be deallocated. We will describe two well known algorithms below. More algorithms exist but we focus on these two. The reason for this is that the mark-and-sweep algorithm is used in the old Boehm implementation, and the copying algorithm is used in the new implementation.

Mark-and-sweep A mark-and-sweep collector algorithm has two phases. In the first phase, the algorithm traverses the graph of live pointers, starting with the root set. All reachable objects are marked as being in use. In the second phase, the heap is linearly scanned from begin to end. All objects that are not marked are considered garbage. The unmarked objects are freed. After the two phases are complete, all garbage is freed and so the memory can be reused by the allocator. Live nodes are left untouched.

Copying The copying collector uses two heaps of the same size, but uses only one at the same time. The collector algorithm also has two phases. When a collection is triggered, it switches the active heap. In the first phase, the root set is copied to the new heap. In the second phase, the new heap is scanned and all nodes referred to from nodes in the new heap are also copied to the new heap. To prevent shared values to be copied multiple times, a forwarding pointer is left at the old location to identify the value is already copied and to share the new location. The second phase is iterated until the new heap is fully scanned. When the two phases are finished, all reachable nodes are copied to the new heap, and all pointers are updated with the new location. The garbage is left in the old heap, so the old heap can be cleared.

Mark-and-sweep versus Copying

Moving memory may seems costly at first, but it has some performance advantages for allocating new memory. By moving all used memory to a new location every garbage collection cycle, usually clears a large continuous region in memory. This new region can be used for allocating new objects without the extra cost of checking if the node will fit. All nodes are compacted every garbage collection cycle. So the only check needed is for enough space. In other words, copying garbage collection prevents memory fragmentation. This results in very cheap memory allocating. And allocation memory is done much more often then running a collection.

A disadvantage of copying collection is that the available memory is divided in in two spaces, and you can only use one space. This divides the available memory in two.

Another disadvantage for copying collection is that it needs cooperation with the mutator. Because all pointers are changed because the objects are copied, we must be sure about what pointers are. This makes copying collection unsuitable for conservative tracing. Mark-and-sweep collection can be used with conservative tracing. This is the reason why the Boehm library uses a mark-and-sweep collector. This is also the reason the Boehm library is used for garbage collection in previous UHC LLVM backend.

2.3 UHC garbage collection library

The UHC compiler provides a garbage collection library. This library provides interfaces for abstraction over garbage collection algorithms. It currently implements a accurate copying garbage collector. To extend the library with support for different backends, a plug-in can be added. To be able to use this library with a specific backend, 3 different types of plug-ins can be implemented. These plug-ins will be used by the garbage collector itself. It defines the way in which the garbage collector can cooperate with the mutator. Beside these plug-ins the garbage collector also defines an interface with garbage specific functions. In this way the mutator can communicate with the garbage collector.

2.3.1 Garbage collector interface

The UHC garbage collector defines a set of functions that can be used by the mutator. Most functions are

alloc(size) This function allocates a specific amount of memory. It returns a pointer to the allocated memory. A garbage collection could be triggered if there is not enough space left.

allocEnsure(size) This function ensures there will be enough space left for the requested amount. If there is not enough space left, it will trigger a collection. Note that no real allocation is done here. This function can be used on garbage collection safe points in combination with allocEnsured, to prevent garbage collection to happen on unsafe moments.

allocEnsured(size) This function allocates memory without checking if there is enough space left. It need to be used in combination with the allocEnsured function.

allocResident(size) This function allocates a piece of memory that will never be freed by the garbage collector. The memory can be scanned for pointers.

allocDeallocResident(pointer) This function can be used to deallocate resident memory. Because it will never be freed by the garbage collector.

registerGCRoot(pointer) This function can be used to inform the garbage collector about a global root value.

registerGCRoots(pointer, count) This function is the same as registerGCRoot, but you can pass more roots at the same time. Useful when having arrays of root pointers.

gc() This function manually triggers a garbage collection cycle.

init() and exit() These function must be called by the mutator to initialize and shutdown the garbage collector.

2.3.2 Plug-in for supplying roots

All compiler backends are different, generate different code, and have a different way of memory usage. The garbage collector needs to be able to find the root set when it wants to perform a garbage collection cycle. This plug-in must enumerate all garbage collection roots at a given moment. As explained in section 2.2.1, roots can be found in different places in memory. Therefore more than one implementation could be present for a specific backend. A good examples for this plug-ins whould be a plug-in that is able to find all roots on the stack.

2.3.3 Plug-in for tracing nodes

This plug-in must provide an implementation for a part of the tracer component, explained in section 2.2.4.The tracer must follow the graph of all live pointers. The algorithm for this can be implemented in different ways. The simplest and most clear way is to do recursive calls when we need to scan another node. But the disadvantage is that we build up a large call stack for every recursive call. This will cost a lot of stack space and is therefore not the solution with the best performance. There is also a possibility of stack overflow when the graph of live pointers is very large. A beter solution could be an iterative approach. A good example of this is Cheney's algorithm [Che70]. This breadth-first algorithm only needs a small amount of memory to keep the state of the trace.

The state of the algorithm can also be managed by a scheduler. The current implementation of the UHC garbage collection library takes this approach. The library abstracts over the way the collection algorithm is implemented. This can be done in a nice way using the scheduling approach. The implementation can be changed without the need of rewriting the tracing plug-ins. The plug-in for tracing nodes must implement a function that can perform a trace on only a single node. If we encounter another pointer we need to follow, we must do a callback to notify the library to schedule the trace for this new node.

2.4 Observations and conclusions

Replacing the conservative library with an accurate garbage collection library will gives us following advantages:

- Better performance of generated executables because the collection cycle is faster.
- Correctness will prevent undesirable behavior and crashes.

We must change the LLVM backend and garbage collection library:

- We must provide the garbage collection library with all global roots
- We must provide the garbage collection library with a routine to find all stack roots
- We must provide the garbage collection library with a routine to find all roots inside objects

In order to make this possible we need to write plug-ins for the UHC garbage collection library.

To be make these plug-ins possible we need to change the LLVM backend. In the next chapter we will explore how the LLVM framework can help with implementing a compiler backend with support for accurate garbage collection.

Chapter 3

The LLVM Framework

LLVM can help a compiler with generating machine specific code. A compiler can output LLVM instructions, and the LLVM framework can help to create optimized native machine code. We will first take a look at the LLVM instruction language itself. Then we investigate the garbage collection support the LLVM framework provides.

3.1 The LLVM instruction set

The LLVM virtual instruction set is designed as a low-level language with high-level type information. It provides language independent type information about all values in the program. The language also exposes memory allocation directly to the compiler. The instruction set represents a virtual architecture that captures the key operations of ordinary processors but avoids machine specific constraints such as physical registers, calling conventions, etc. LLVM provides an unbound set of typed virtual registers which can hold values of primitive types. Examples of primitive types are integers and pointers. Assignments are in Static Single Assignment (SSA) form. SSA form is a widely used style in which virtual registers are assigned exactly once. SSA simplifies the properties of variables thereby simplifying and improving the results of a variety of compiler optimizations. LLVM combines a register based machine with a stack based machine.

Memory is partitioned into a global area, stack, and heap. Objects on the stack are allocated using the alloca instruction, and objects on the heap are allocated using the malloc instruction. These instructions return pointers that can be used to access the memory. Stack objects are allocated in the stack frame of the current function. Heap objects must be explicitly freed using a free instruction. The alloca instruction is similar to malloc except that it allocates memory in the stack frame of the current function instead of the heap, and the memory is automatically deallocated on return from the function. This behavior is similar to what we expect from the ideas behind the memory areas explained in section 2.1.

3.1.1 Assignments

The LLVM language contains instructions to manipulate memory. Let's take a look at some example code that shows how assignments are done. See code listings 3.1 and 3.2. We compare a small piece of C code with the corresponding LLVM code.

Listing 3.1: C assignment example

```
// initializing x
int x = 5;

// incrementing x
x++;
```

Listing 3.2: LLVM assignment example

```
; initializing x
%x   = alloca int32
store i32 5, i32* %x

; incrementing x
%x2  = load i32* %x
%res = add i32 %x2, i32 1
store i32 %res, i32* %x
```

- First we declare a integer variable x and initialize it with value 5. In C this can be done with a one-liner. For LLVM this requires two steps. First we need to reserve stack space where we can store the value of x later. Note that we actually have a choice here. We could also store it on the heap. The stack allocation gives us a integer pointer we can use to read and write to this memory location. The second line of code stores a integer with value 5 at the reserved position on the stack. Note that the C code wil probably also store the value of x on the stack, depending on the compiler, but this is done implicitly.
- The second example is to increment the value. In C we can use the increment operator. Note that this operator is short for "x = x + 1". For LLVM this means we have to do 3 steps. First we have to load the original value of x. Second step is to add 1 to x. Note that we need two different virtual registers to store the intermediate results. Virtual registers are sufficient for x2 and res, but for x we really need stack space because we want to mutate the value. The final step is to store the result in memory on the stack location reserved for x.

Note that all LLVM instructions are annotated with high level type information. This type information enables high-level transformations on instructions that are actually low-level. Also note that when we compare LLVM instructions with low level assembly instructions, we should expect that we would also be required to clear the stack space. This is not necessary because of the way LLVM handles functions. We explain this in next section.

3.1.2 Function calls

LLVM does aggressive link- and post-link time optimizations. Therefore high level function calls are supported using build in calling conventions. In listing 3.4 we can see an example function call. Standard calling conventions, like the C calling convention, can be used, or a custom calling convention can be added. LLVM functions also automatically clear all local variables on the stack. In listing 3.4 we can see an example function call. Support for exceptions is also available, hence the nounwind in the LLVM example, but we do not use this so we will not further explain this.

When real machine instructions are generated using a stack memory model, a lot of administration has to take place to be able to call a function. For example these steps have to be performed to call a function with parameters:

- Push function parameters on the stack
- Jump to the function code to execute the function
- Remove parameters from the stack when the function ends

Listing 3.3: C function example

```
// function call
int x = f(2);

// function definition
int f(int p){
    return 3;
}
```

Listing 3.4: LLVM function example

```
; function call
%x = call i32 @f( i32 2 )

; function definition
define i32 void @f( int32 %p
) nounwind {
    ret i32 3
}
```

- Remove local variables from the stack when the function ends
- Push the result of the function on the stack.
- Jump to the return address.

LLVM abstracts from function calling, simplifying code generation and making program analysis easier.

3.1.3 Heap allocations

In the beginning of this section we said that LLVM has a separate heap. We can use it by calling the malloc instruction as shown in example 3.5. Heap space always has to be freed explicitly. This can be done by using the free instruction. Using the build in heap from LLVM is not necessary. When garbage collection is added one might want to implement a separate heap for more control. For this reason the malloc and free instructions are removed from LLVM recently. You can use the malloc and free from the standard C library.

Listing 3.5: LLVM heap allocation example

```
; Declaring heap space
%y = malloc i32

; Storing a value to the heap
store i32 5, i32* %y

; Deallocate the heap space
free int32 %y
```

3.2 LLVM optimizations

The LLVM framework features a collection of transformation and analysis passes, some performing optimizations. LLVM code is written in SSA form. This enables or strongly enhances many optimizations. The optimizations are performed in two stages of the compiler. The first class of optimizations are performing LLVM to LLVM transformations, including typical SSA optimizations like dead code elimination. The optimizations are performed by the opt tool provided by the LLVM framework. The user is free to select what optimizations need to be performed while compiling, or a default set of optimizations can be chosen. The result of this stage is an optimized LLVM bytecode file. Examples of optimizations the LLVM optimizer is able to do are:

- Dead code, argument or instruction elimination.
- Constant propagation.
- Function integration/inlining.
- Partial function specialization.

The second stage of the compiler is low level code generation. Besides generating a native binary, also a JIT compiler could be used. The LLVM framework has a large collection of backends specialized in generating code for many machine architectures including X86, PowerPC, Alpha, and SPARC. During low level code generation other optimizations are performed. Important optimizations are register optimizations. Registers can be utilized to reduce memory access time. An advantage of programs written in SSA form is that they can be colored in polynomial time. Graph coloring is a technique for efficiently using the limited set of CPU registers.

This results in optimized programs that will probably perform better. The optimizer can for example decide to omit memory allocations or use one memory location for multiple values that are used after each other. Other values might be written to registers instead of memory. So a lot of things can happen that interfere with our initial idea of how memory is used. This might be a problem because we want to use this initial idea for the garbage collector to describe in memory location of the root set. If the root set description has become invalid after optimization, it can break a garbage collector.

A solution for this could be making the LLVM optimizer aware of which references we want to keep track of for later use by the garbage collection. The optimizer could track these garbage collection annotated references and tell us where they are at a specific time. Then we still have a valid root set so we can do accurate garbage collection. The LLVM optimizer is currently not able to preserve garbage collector information. It should be possible to make it aware but it currently is not. So we have to consider LLVM compiler as a hostile environment, like compiling to C code and generating native assembly by GCC. Another solution for managing garbage collection roots has to be found to make accurate garbage collection possible.

3.3 The shadow-stack

Many compilers choose not to compile to native bytecode, but try to compile to an intermediate or high level language, like C or LLVM. Reason for this is that other compilers for these languages exist and the implementor bothered with details of different machine platforms. The backend that already exists can simply be reused. Implementing accurate garbage collection for these backends can be difficult if they do not support it. The garbage collector will have difficulties to trace the stack because it is controlled by the backend we have no influence on. A solution for this is to use conservative garbage collection, but this solution has some drawbacks as explained in chapter 2.2.4. Another solution is to completely avoid the native stack used by the backend, and use an own virtual stack. For LLVM this would mean that we do not use the alloca instructions, but reserve a piece of heap memory and use it as a stack. We can create our own stack descriptor and it will not get broken because the compiler will not change the stack layout of our own virtual stack. But this approach eliminates many advantages of using the native stack of an existing backend. A lot of machinery has to be reimplemented for the used backend, and some opportunities for optimization are lost.

An alternative approach is using the shadow-stack transformation. This is an alternative approach that allows fully type-accurate and liveness-accurate garbage collection, thus allowing garbage collection techniques that would normally require support from the backend [Hen02]. Henderson describes this technique as a transformation on generated C code, but it can be done for LLVM as well. The LLVM framework already contains a compiler pass that is able to do this transformation. The LLVM documentation suggest you use the shadow-stack for accurate garbage collection, because it makes your garbage collection entirely independent from the LLVM backend optimizations.

3.3.1 The LLVM code transformation

Some compilers want to benefit from accurate garbage collection. All steps in the compiler pipeline must have support for garbage collection to be able to do accurate garbage collection. Recall that accurate garbage collection needs cooperation with the target environment. The LLVM framework supplies basic mechanisms and interfaces to support accurate garbage collection. The goal of the LLVM framework is to support various types of advanced garbage collectors models like generational and concurrent collectors. The LLVM framework does not provide actual implementations for these garbage collection algorithms.

The shadow-stack transformation on LLVM uses the same idea as the C transformation described by [Hen02]. The only difference is that instead of tagless approach a meta table approach is used. the function pointer is replaced with a data table. This allows more flexibility: the garbage collector implementer can implement its own tracing functions. We will look deeper into the technical details of this implementation because we really need to understand what is going on to be able to implement garbage collection on top of it.

The shadow-stack transformation transforms the code to maintain the shadow-stack datatype. The shadow-stack datatype is a dynamic structure that is maintained on the LLVM stack. Normally garbage collection roots are scattered on the stack and we are not able to point at them. But the shadow-stack is a linked list that points to all the roots. The roots are grouped in an array and described by a frame map. The frame map contains information about the roots and contains a pointer to the next frame map. Every function call adds a group of garbage collection roots to the linked list, unless the function does not have roots.

A global variable is maintained that points to the last frame map. If the garbage collector wants to walk the stack to process all the roots on the stack, it has to follow the linked list. The shadow-stack allows the garbage collector to find all the roots at runtime.

Compiler plug-in

To be able to use the shadow stack with LLVM, an annotation can be used to enable a compiler plug-in. A compiler plug-in can be used to define an extra step in the compilation process. For example it can be used to do LLVM to LLVM code transformations. This is also what the compiler plug-in for the shadow stack does. It performs the source-to-source transformation similar to the C code transformation explained in previous section. A compiler plug-in can be selected by using the "gc" language feature companied with the plug-in name. For example if we want to use the shadow stack plug-in, we accompany all functions with gc "shadow-stack".

Intrinsics

Besides LLVM instructions, LLVM also supports the notion of an "intrinsic function". These are special functions used for extensions of the language. The meaning of the intrinsic is defined by extensions of the compiler.

The LLVM framework also defines a garbage collection intrinsic. References that are maintained by the garbage collector can be identified by using intrinsics. The intrinsic itself means nothing to the code, it is only an annotation. The actual meaning of the intrinsic is specified by the compiler plug-in. The shadow-stack compiler plug-in looks for these intrinsics and uses them for the transformation to maintain the shadow-stack. So we need to indicate the roots to help the shadow-stack transformation.

Listing 3.6: gcroot intrinsic

```
declare void @llvm.gcroot(i8** %ptrloc, i8* %metadata)
```

The first argument must be a value referring to an alloca instruction. So a local variable placed on the stack. It could also be a bitcast of the stack variable pointer. Because the argument type is i8, sometimes a bitcast is needed to make the type checker happy. The second contains a pointer to metadata that should be associated with the root, and must be a constant or global value address. It could also be a pointer to a function for tag free garbage collection. Tag free garbage collection is further explained in section 6.1.1. If your target collector uses tags, a null pointer can be used. No extra stack space is needed if the meta data is null.

LLVM shadow-stack example

We take a look at the shadow-stack and gcroot intrinsic in large example. Figure 3.8 shows the LLVM code. Next to the code is a snap shot of the stack. We simulate that fun1 is called, and fun1 calls fun2. The snapshot of the stack is taken right before fun2 ends. We will now explain what is happening chronologically.

1. First we reserve stack space for two locals. These two local will contain pointers maintained by garbage collection.
2. Also reserve stack space for a local that is not managed by the garbage collector. Also store a simple integer value at the stack. Note that beside the types, there is no real difference yet between all locals.
3. Reserve heap space for the two garbage collected locals. We here use the build in malloc function for simplicity, but usually an external malloc function will be used when implementing a garbage collector. The pointers return by malloc are stored at the reserved stack positions.
4. Now we call the gcroot intrinsics. Before we can do, we first have to cast the 32 bit pointer to an 8bit pointer. This is necessary to make the type checker happy. Then we do the actual call. Note that local_2 is tagged with metadata put local_1 is not. The descr value is a pointer to a global defined somewhere in the program but we omit the definition.
5. Now we call func2.
6. Fun2 is a simple small function with just one local maintained by garbage collection. All concepts used here are already explained. This function is added only to show how function calls will influence the stack.

We will now take a closer look at the picture to see how the stack looks like at runtime. The datatypes LLVM uses for maintaining the shadow-stack are shown in figure 3.7. This could be used as reference to better understand the picture. For every function call a specific region is distinguishable where all data for that specific function is stored. This region is called a function frame. Data typically found there are locals, parameters, etc. But also the dynamic structure created for the shadow-stack is located here. We can see that for every function call a StackEntry is created. This structure is initialized immediately when the function is called. It contains a pointer to the next function frame. This is the previous function call that is not yet returned. The StackEntry contains a FrameMap. This FrameMap stores the ammount of gcroots and meta descriptions that are stored within the StackEntry. Note that this not needs to be the same because not every root has a meta description attached. All gcroots are stored separately in the StackEntry. The reason why they are stored separately is that no useless space is needed for roots that have no meta information attached.

Listing 3.7: LLVM Shadow stack datatypes

```
1
2  struct FrameMap
3  {
4      int32_t NumRoots;          //< Number of roots in stack frame.
5      int32_t NumMeta;           //< Number of metadata entries. May be < NumRoots.
6      const void *Meta[0];       //< Metadata for each root.
7  };
8
9  struct StackEntry
10 {
11     struct StackEntry *Next;   //< Link to next stack entry (the caller's).
12     const struct FrameMap *Map; //< Pointer to constant FrameMap.
13     void *Roots[0];            //< Stack roots (in-place array).
```

```
14 };
15
16 struct StackEntry *llvm_gc_root_chain;
```

The picture shows the situation for our example code almost before fun2 has ended. We can find both gcroots of fun1 stored in the function frame for fun1. We only added meta information for 1 gcroot. The ordering of the gcroots depends on whether meta information is added. First all gcroots with meta information are listed, so both list can be iterated side by side. This is the reason that local_2 is listed before local_1. The order of the roots is not important because the roots are identified by using either the meta information or some tag within the data itself.

We can use this dynamic data structure that is created on the stack to write a algorithm that walks the stack en visits all garbage collection roots on the stack. The roots can be identified by looking up the accompanied meta information. Or the garbage collector can dynamically inspect the heap where it is pointing to.

3.3.2 Correctness

The shadow-stack transformation makes it possible to do accurate garbage collection without support from the backend. This means that the backend can not break the extra descriptive information needed for garbage collection. But how is this possible? The answer is simple, everything is implemented in normal code. The compiler must preserve the semantics of the program. So the information used for the stack descriptor is also preserved. The compiler can not do any unsafe optimizations. After the transformation, local variables managed by the garbage collector are actually used two fold: for the program itself and for garbage collector.

3.3.3 Disadvantages

In previous section we stated that the shadow-stack transformation prevents the compiler from doing any unsafe optimizations. But this also has a downside. Inhibiting such optimizations eliminates some opportunities for getting better performance. Garbage collection and LLVM optimization are actually conflicting. Some optimizations are still possible like function inlining. The frame descriptors do not have any relations with stack frames. Also local variables not managed by the stack descriptors can be freely optimized by the backend compiler.

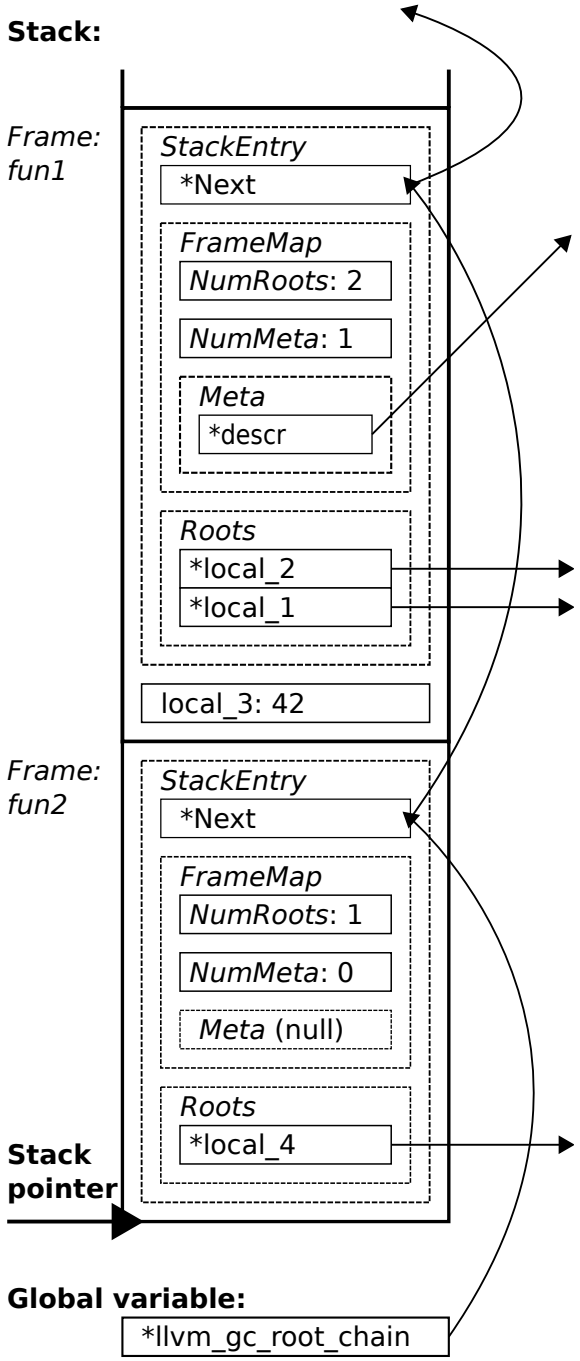
But some disadvantages for optimization remain:

- The shadow-stack transformation eliminates an important optimization. Many compilers optimize non-inlined function calls by passing arguments through registers. This can have a significant impact on performance. Boquist describes register optimizations as one of the most important optimizations [Boq99]. So the idea of shadow-stack also conflicts with the GRIN optimizations.
- Another disadvantage is the stack overhead needed for keeping the chain administration. This costs extra stack space for storing the administration. Also extra instructions for the mutator are needed to update this administration.

The shadow-stack transformation is not garbage collection with no overhead. Looking at these disadvantages, if we choose to use the LLVM framework we should have to measure and analyze the performance results. This raises another important research question: "does the shadow-stack transformation results in acceptable overhead?". We need empirical evidence to support this claim. Functional languages like Haskell are known to do many function calls. It is important that function calls are made cheap for compilers of functional languages.

Listing 3.8: LLVM function example

```
define void @fun1() nounwind gc "shadow-stack"
{
    ;;;; Step 1.
    %local_1 = alloca i32*, i32 1
    %local_2 = alloca i32*, i32 1

    ;;;; Step 2.
    %local_3 = alloca i32, i32 1
    store i32 42, i32* %local_3

    ;;;; Step 3.
    %heap_pointer_1 = malloc i32
    store i32* %heap_pointer_1,
          i32** %local_1

    %heap_pointer_2 = malloc i32
    store i32* %heap_pointer_2,
          i32** %local_2

    ;;;; Step 4.
    %local_1_cast = bitcast i32** %local_1
                          to i8**
    call void @llvm.gcroot(i8** %local_1_cast,
                           i8*  null )

    %local_2_cast = bitcast i32** %local_2
                          to i8**
    call void @llvm.gcroot(i8** %local_2_cast,
                           i8*  %descr )
    ;;;; Step 5.
    call void @fun2()
}
define void @fun2() nounwind gc "shadow-stack"
{
    ;;;; Step 6.
    %local_4 = alloca i32, i32 1

    %heap_pointer = malloc i32
    store i32* %heap_pointer, i32** %local_4

    %local_4_cast = bitcast i32** %local_4
                        to i8**
    call void @llvm.gcroot(i8** %local_4_cast,
                           i8*  null )

    ;;;; From this point the
    ;;;; stack picture is shown
}
```



3.4 Read and Write bariers

Some garbage collectors require special code to be executed when values are written to specific memory locations, like generational garbage collectors. LLVM provides intrinsics to identify these memory writes. LLVM variables can be annotated with the gcwrite intrinsic. The signature of this function is shown is figure 3.9. By default, the semantics of the gcwrite intrinsic is the same as a store. But the semantics can be changed by implementing a compiler plug-in. Example of desired functionality could be a call to the garbage collection library to inform a reference is been written to memory. This could be needed for updating the remembered set for generational garbage collection. The intrinsic may only be used in a function when a garbage collection algorithm is specified.

The store instruction has, besides the type argument, two arguments. First argument is the value to store, the second argument is the address at which to store it. The contents of memory are updated to contain the value of the first argument at the location specified by the second pointer argument. The gcwrite intrinsic is similar, but a little more restrictive and specialized for use with complexer LLVM datatypes. Like the store instruction, the first argument is the value to store, but it should always be a reference. The second argument is the address at which to store it, but we can also give a third argument for calculating an optional field at the object this reference points to. This behavior is similar as the "getelementptr" instruction. This instruction can calculate memory offsets for complex LLVM types given a start address of this complex object. LLVM expects us to create complex LLVM datatypes for garbage collected values. But this is not necessary.

Listing 3.9: gcwrite intrinsic and store instruction

```
; signature
declare void @llvm.gcwrite(i8* %P1, i8* %Obj, i8** %P2)

; definition
store <ty> <value>, <ty>* <pointer>
```

Another garbage collectors require special code to be executed when values are read from specific memory locations, like concurrent garbage collectors. LLVM provides intrinsics to identify these memory read. LLVM variables can be annotated with the gcwrite intrinsic. The signature of this function is shown is figure 3.10. By default, the semantics of the gcread intrinsic is the same as a load. But the semantics can be changed by implementing a compiler plug-in. A read barrier could be used for protecting writes to an object when implementing a concurrent garbage collector. We do not focus on concurrent collection, so we do not go into detail.

Listing 3.10: gcread intrinsic and load instruction

```
; signature
declare i8* @llvm.gcread(i8* %ObjPtr, i8** %Ptr)

; definition
<result> = load <ty>* <pointer>
```

3.5 Related work

LLVM can be used as intermediate language to abastract over low level details. LLVM has garbage collection support as we explained in this chapter. We will now take a look at different low level frameworks that also provide support for garbage collection.

3.5.1 Managed languages

High level languages like Java and C# also compile to a low level language comparable to LLVM. Java compiles to Java bytecode and C# compiles to MSIL [ECM06]. Java bytecode, MSIL instructions and LLVM are low level program representations but abstract over machine specific details. The most important reason for this is to work towards portability. To be able to run the compiled applications, LLVM instructions have to be compiled to a native executable. Java and C# instructions can be interpret by a virtual machine that really executes the code. Recently also JIT compilers are available for all three languages. JIT compilers can compile the abstract bytecode on-the-fly at runtime to native machine code and directly execute it.

To this far LLVM, Java and MSIL are really comparable to each other. When it comes to memory management and garbage collection, the languages are different. The Java and MSIL frameworks have complete build in support for garbage collection. The runtimes come with implementations for garbage collection algorithms. Garbage collection works out-of-the-box on all compiled Java or C# programs. This is caused by the well defined binary interface of Java and C# memory objects. This binary interface is suited for the objects used by this object oriented language. The garbage collection works because it knows how to handle memory objects that follow this standard.

The LLVM framework does not forces an in memory object representation. Therefore the LLVM framework does not come with an implementation of a garbage collector. The algorithm has to be tailored for the specific in memory object representation used by the implementer. LLVM can not decide on this but it also gives the implementor more freedom. The support for garbage collection in LLVM is of a different class. It helps the implementor only with non-trivial tasks like finding garbage collection roots on the stack. Therefor we could say that the garbage collection support for LLVM plays a smaller role compared to the support for gc in other abstraction languages like Java and C#. But this is caused by the choice the LLVM developers have made. Their philosophy is to provide a low level framework not bounded to a specific programming style like object orientation.

The differences can be recognized in the languages itself. Price to pay for out-of-the-box garbage collection is that the Java and MSIL language do not have a way to manually delete objects. On the other hand creation of object is of concern of the implementor. Also there is no way to manually trigger garbage collection.

3.6 Observations and conclusions

We have now seen how the LLVM framework can help us with implementing accurate garbage collection. So we can now answer the question: "How can the LLVM framework aid us with accurate garbage collection for a compiler backend?" by summarizing the important aspects of this chapter:

The LLVM framework can help us with following tasks:

- Creating a stack map descriptor.
- Writing specific code for read and write barriers.

If we use the intrinsics interface the underlying implementation can change in a newer version the shadow-stack to a static stack map without changes to our backend an making our backend automatically work (and faster)

The LLVM framework cannot help us with:

- Providing a garbage collection algorithm implementation.
- LLVM does not suggest an binary interface for in memory object representations.

- Registering global roots. It only has support for roots on the stack.

So we need to address these issues ourself. LLVM framework gives a lot of freedom concerning these aspects.

Also LLVM garbage collection support is no golden bullet.

- Shadow-stack has runtime overhead.
- Shadow-stack might miss optimization opportunities.

We now know how the LLVM Framework can help us with garbage collection. In next chapter we will describe how we make use of this in describing the implementation of the LLVM backend.

Chapter 4

Implementation

In chapter 3 we have looked at the functionality the LLVM framework provides for implementing garbage collection. We also investigated the needs of UHC garbage collector in chapter 2. We want to add accurate garbage collection to the LLVM backend. The UHC runtime already contains an implementation of a heap memory allocator and a garbage collection algorithm. The UHC garbage collector is able to perform accurate garbage collection. The software engineering job that needs to be done is connecting the UHC LLVM backend with the UHC garbage collection library. We will investigate what modifications and extensions we need to make this possible.

From a high level perspective we have two major tasks:

- We need to change the LLVM backend to emit code that can cooperate with the garbage collector.
- We need to add functionality to the UHC garbage collector so it can cooperate with the LLVM backend.

Two software components are important: The UHC LLVM backend and the UHC garbage collection library. The two task will grow these two components together to make the accurate garbage collection work.

4.1 Extending the LLVM backend

- Replacing the current Boehm heap allocator with the UHC heap allocator.
- Adding functionality for the garbage collector to find all garbage collection roots.
- Generating node descriptors that can be used for tracing.

Replacing heap allocation calls is a trivial task. We can replace the already existing calls to the Boehm allocator with calls to the UHC allocator. No further changes are needed. Heap memory is now in control of the UHC garbage collector. The heap needs to be under control of the garbage collector.

The garbage collector must be able to find all garbage collection roots. Garbage collection roots can be found in different memory locations. Places to look for garbage collections roots are global memory allocations, the function return area and the stack. This concept is further explained in section 4.1.1.

We must be able to generate node descriptors. We need to investigate all possible values that could be written to heap locations. And for all different values we need to have some sort of descriptor that describes this value in a way the garbage collector can handle it. This concept is further explained in section 4.1.2.

4.1.1 Finding garbage collection roots

Roots are starting point for performing a garbage collection cycle as explained in chapter 2. Root pointers can be found at different places in memory. We first describe all possible memory locations.

The result pointer (RP)

An LLVM module always contains a global variable used for returning grin nodes from functions. This special global is called RP. RP stands for Result Pointer. This RP is needed because LLVM does not allow functions to have multiple result values. Complex grin nodes can be returned by LLVM functions by storing the value in the RP. Enough memory is permanently allocated for RP to store any grin value possible to be returned by functions for the specific program. The characteristics of the RP are similar to other globals. The memory needed for the RP itself does not need to be collected for the entire lifetime of the program. But the RP will contain grin nodes. RP does not need to be traced because pointers in the RP will already become roots of the function using it. The grin node in the RP is copied to be a local of the function using it. No garbage collection could happen in between because no memory allocation is done between returning a function and copying the result to a local. The RP does not require any changes to make accurate garbage collection possible.

Globals

A LLVM module can contain global variables. These global variables exist to share top level values like constants. The memory used for global variables does not need to be garbage collected because the lifetime of a global is as long as the mutator is executing. Globals are therefore allocated permanently on the heap. But globals will contain grin nodes. And grin nodes can contain pointers to other heap nodes. We need to treat these globals as garbage collection roots, because otherwise heap cells only pointed by globals could mistakenly become garbage, and the memory will be lost during a garbage collection cycle. This causes dangling references and can cause the mutator to crash. Therefore all globals must be treated as roots by the garbage collector. The only difference with normal heap cells is that the space for globals never needs to be deallocated themselves. Therefor a special root supply plug-in for globals is created, that only scans globals but never deallocates them.

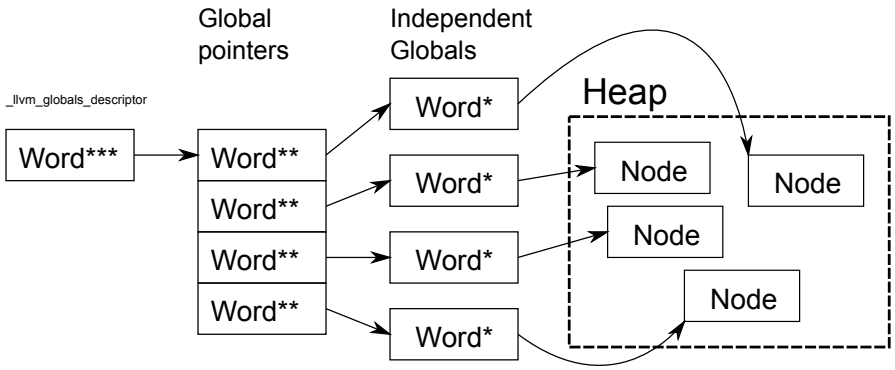


Figure 4.1: Globals descriptor

In the original backend the globals were independent of each other. We introduce an new array with pointers to all program globals. This approach introduces an indirection. We could also put the globales directly in an array. But this has the disadvantage that globals needs to be looked up when they are needed. Introducing a new indrection array does not change much for the mutator, and the indirections are only used once when starting a garbage collection. So this is acceptable. This array can be seen in figure 4.1. The array is generated for every llvm module. It is initialized with pointers to all independent globals. Now the garbage collector plug-in can use this array to find all globals.

The stack

To be able to find the local variables on the stack that contain pointers which are under control of garbage collection, we use the shadow-stack as explained in chapter 3. All local variables that can contain pointers managed by the garbage collector are registered with the gcroot intrinsic. Now we are able to find these pointers at runtime when we want to perform garbage collection. And example of allocating a grin node under control of garbage collection is shown in listing 4.1.

The gcroot intrinsic gives the possibility to add meta info with the root. We will not make use of this feature for some reasons explained later in section 4.1.3. So we simply pass a null value as second argument of the gcroot intrinsic.

Listing 4.1: Heap allocation for grin nodes

```
; Allocating heap space using the UHC garbage collection library
%fresh170 = call i64* @mm_itf_alloc_ext( i64 16, i64 0 )

; Store the pointer in a local variable %x51
store i64* %fresh170, i64** %x51

; Bitcast the pointer to fit the interface of gcroot
%fresh171 = bitcast i64** %x51 to i8**

; Call gcroot to put the local containing a pointer that is
; under control by the garbage collector on the shadow stack
call void @llvm.gcroot( i8** %fresh171, i8* null )
```

Function parameters

A LLVM function can have parameters. When an LLVM function is called, paramenters are copied on the stack, and the local copies are used by the function. These parameters can contain pointers. During garbage collection these pointers need to be updated to point to the location in the new heap. LLVM handles argument passing when a function is called. The argument will be placed as a copy on the stack but we have no direct control of it. Therefore we introduce an extra local function variable for every argument that contains a copy of the argument value. We mark this value as gcroot. An example of this can be seen in listing 4.2. Now the garbage collector will reach this pointer during a garbage collection cycle and will update it with the new pointer. When a parameter is needed, the local copy is used in stead of the parameter. This ensures we always get an up to date pointer value that might have been changed during a garbage collection.

Listing 4.2: Function parameters are copied locally

```
define internal void @fun_sum( i64 %x27 ) nounwind gc "shadow-stack"
{
    ;... stack allocations for locals

    %par_x27 = alloca i64*, i32 1
    %cast1_par_x27 = inttoptr i64 %x27 to i64*
    store i64* %cast1_par_x27, i64** %par_x27
    %cast2_par_x27 = bitcast i64** %par_x27 to i8**
    call void @llvm.gcroot( i8** %cast2_par_x27, i8* null )
```

Tag Type	Tag Name	Arity	Primitives	Updatable	Updated with value of type	Max arity
C	Nil	0	[]	No	-	0
C	Cons	2	[No, No]	No	-	2
C	Int	1	[Yes]	No	-	1
P	Upto1	1	[No]	No	-	1
F	Upto	2	[No, No]	Yes	[a]	2
F	Upto1	1	[No]	Yes	[a]	2
A	Map	3	[No, No, No]	Yes	[a]	3
H	Hole	0	[]	Yes	Depends	Depends

Table 4.1: Grin node variations

4.1.2 Grin nodes

The heap will only be used by our mutator for allocating grin nodes. The garbage collector must know the nodes are constructed, to be able to scan and collect them. When compiling grin to silly and eventually LLVM, information about the grin nodes is passed. This information can be used to generate grin node descriptors. Before we can decide how to define these descriptors, we need to take a closer look to how grin nodes are encoded in memory, and at all possible grin node variations that exist. Examples of grin nodes can be seen in table 4.1 The meaning of the columns is discussed in next section.

Grin node encoding

A grin node is represented in memory by a set of fixed size fields. The first field always contains a constructor value, used to identify the grin node type, and the data it contains. The constructor is accompanied by zero or more data fields. The number of fields in use is called its arity. These data fields can contain either a primitive value or a pointer value. Pointers to other grin nodes in memory are used to encode grin level connections. The size of the field should be enough to store a pointer. Therefore the size of fields is chosen to be the machine word size. For example compiling for a 32 bit platform will result in a field sizes of 4 bytes. Besides pointers, machine words can also hold integer values. Also grin tags can be stored in a Word because the tags are encoded as integer values during an earlier step in the compiler pipeline.

Constructor nodes (C tag)

Constructors nodes are simple data containers. Constructor nodes could be recursive or primitive. If the node is recursive it will contain pointers. Constructor nodes are not updateable. Constructor nodes are usually the results of a computation and have their final node type. This means the node type will not be changed and so it will never grow, thus no extra space needs to be reserved for constructor nodes.

Function applications (F tag)

A function application node is used to store suspended function calls. Haskell has a lazy evaluation strategy. This means that functions are only executed when the result is needed. These function application nodes are used to store a function call with its arguments so it can be executed when the results is needed. The tag specifies which function is called, and the datafields contain the values for the function arguments. When the result is needed, the function is executed and the function application node is updated with the result of the function. Storing the function result gives us the advantage that we do not have to compute the result again if the value

is used more then once. This is possible because of the pure nature of Haskell function calls. Pure Haskell functions cannot have side effects so the result of the function will always be the same.

The constructor node resulting from the function application can require more space than originally was needed to just store the function call. Enough space needs to be reserved when the function node is created to be able to store the constructor node in-place later. We choose the simple solution to reserve enough space, another solution could be an overflowing mechanism. To be able to do this, we need to know how much space we will need in the future.

Partial applications (P tag)

Like function applications, partial applications capture free variables. When all free variables are captured, the partial application is abandoned, and a new function application node is created. So partial applications are never updated inplace with a different node. So the problem for updates is not relevant for partial applications.

Applications of higher order functions (A tag)

Higher order applications are treated differently than normal function applications. We need to distinct them from normal function applications to be able to evaluate them. The representation and behavior is not different from normal function applications. So we need to take care of them in the same way as normal function calls.

Holes (H tag)

Holes are special grin nodes. Holes represent reserved space for a grin node to be constructed later. This is sometimes needed for mutually recursive data structures. The tag and data fields for holes are not yet known, but we need to reserve memory for them already to be able to get a reference to it. This reference is used to refer to this node later. The hole will not be used by the mutator until the actual value is stored.

4.1.3 Grin node descriptors

Now we have seen all possible variations of grin nodes, we can decide on how to describe them in a way the garbage collector can interact with them. To recall, the garbage collector has two important concerns:

- The tracer must be able to scan all live heap nodes.
- The collector must be able to reconstruct the memory that is still in use.

Grin node descriptor examples			
Tag Name	Arity	Has pointers?	Max arity
CNil	0	Yes	0
CCons	2	Yes	2
CInt	1	No	1
PUpto1	1	Yes	1
FUpto	2	Yes	2
FUpto1	1	Yes	2
AMap	3	Yes	3
Hole	0	Depends	Depends

Table 4.2: Grin node variations

Observations

To able to trace reachable memory, it is important to know what data fields of grin nodes contain pointers. If we know where to find pointers, we can follow them and reach all live memory. For every grin node the information about pointers is available at compile time. So we build a lookup table that tells us if a field contains a pointer or a primitive value to make this information available for the garbage collector at runtime. An example table is shown at 4.2. We will now explain our choices.

Currently almost all grin nodes contain pointers because we only use boxed datatypes. There are only a few constructor nodes that contain primitives, for example Char and Int nodes. So this allows us ton define per tag if it contains a primitive value or not. If the node contains a primitive value, it is always one field. So this makes the situation very simple and we can deal with just a simple boolean. We store this information in the "has pointers?" column. When unboxed values are being implemented this simple boolean needs to be extended with per field information. But we can not predict how unboxed values are going to be implemented so this solution is adequate for now. More information about unboxed datatypes is available in section 6.2.1.

To be able to reconstruct the grin nodes, we need to know how much space we need to allocated. The information is available at compile time, but the decision has to be made at runtime. Simply looking at the tag does not gives us enough information. As explained earlier, nodes can be updated with larger values. To be able to construct the new memory during garbage collection, we need to know the maximum size. At compile time we have access to this information. The information can be extracted from the HPT analysis result [Boq99]. HPT analysis is a dataflow analysis that is performed in an early stage of the compiler and is used by various optimizations. We need to store the max arity of a specific tag in the lookup table also so we can always access this information. We store this information in the "max arity" column.

In our node descriptor table we put all information the garbage collector needs to be able to perform a garbage collection cycle. In previous section we analyzed that we need to know current arity, max arity, and primitive fields. Figure 4.2 shows how this descriptor table would look like for previous grin node variations example shown in figure 4.1 .

Forwarding pointers and Holes

The two space copying garbage collector algorithm makes use of forwarding pointers to prevent shared nodes to be copied multiple times, as explained in section 2.2.5. We need to identify forwarding pointers. In section 4.1.2 is explained that tags are encoded as integers. So we can reserve one tag value to identify a forwarding tag. We use the highest possible int value for this. Beside the identification tag we also need to store a pointer to the new location for the node. This pointer can be stored in the first field of the node. We must ensure that every node has at least memory space for two fields, a tag and a forwarding pointer. This is an important requirement.

The only problem still remains is maximum arity for Holes. When the collector encounters a hole, it cannot look up the value for max arity. Because a hole node can be filled with any node. So a naive approach would be the maximum value of all maximum arities. This value is a bit pessimistic. We could find the max arity for a specific instance of a hole by looking at the HPT table. In the HPT table can be found what specific nodes can be stored in the hole. We can locally store this arity value in the second field of the grin node. The garbage collector can then access this information during a collection. The minimum field size of grin nodes is currently two because of the need for forwarding pointers. So we could benefit from this extra space for the max arity of the holes when needed.

Implementation

The table as shown in figure 4.2 can be implemented by using an array of structs. The index of the array represents the tag. This is possible since tags are encoded by integers at silly level. We create a struct that holds the values for arity, has pointers and max arity.

Tagging stack roots using llvm

In section 3.3.1 we explained that we could benefit from the ability LLVM provides us for tagging stacks roots with meta information. This could be an alternative for the tagged approach we currently have chosen. A stack root can be paired with a pointer. At first this might look interesting. If we store a pointer directly to the entry in the node descriptor table this safes us from the costs for looking up this value in the table. But there are two reasons why we do not use this.

- Heap nodes could be updated with a different tag and content. If a stack root with meta data exist that is pointing to this node, the tagged pointer becomes invalid, because it is still pointing to the old location in the table. We do not always access the heap nodes trough a stack root, so we could not always easily change the paired meta pointer. We could try to solve this problem by manually traversing the shadow-stack and update all relevant meta pointers. But this is not a good idea because it to expensive to do this for all tag updates.
- Solving the problem of finding the correct node descriptor for stack roots only is not a complete solution. We also must be able to find the correct description table entry by analyzing the node itself, during the second heap scanning phase of the garbage collector. So we need a mechanism to find the correct descriptor given a specific tag anyway. So we can take the same approach of dynamically inspecting for the stack roots as well.

Using the meta data field of the LLVM shadow-stack will be more usable for the implementation of languages of which the heap data objects do not significantly change during the execution of the program.

4.1.4 Extending the abstract systax

The abstract syntax of LLVM of the current implementation is created on demand. This means that not every possible LLVM construct could be expressed by the current abstract syntax. We performed the following changes to extend the LLVM backend with accurate garbage collection using LLVM:

- Support for "gc" specification language feature. This feature is discussed in section 3.3.1. The implementation makes use of this to enable the "shadow-stack" LLVM compiler plug-in to emit suitable code to maintain the shadow-stack.
- Support for derived and aggregate types. LLVM supports complex types like arrays and structs. Aggregate types are a subset of derived types that can contain multiple primitive types. These complex types are used to build the grin node descriptor table. The types are shared with the C runtime, so we have a common data representation.

For all features listed above we extended the abstract syntax. Also pretty printers were added. Current code is updated to cooperate with the new features.

4.1.5 Grin-to-silly phase modifications

Some required changes for garbage collection to work could be implemented by changing the grin-to-silly phase more easily than implementing it directly into the silly-to-LLVM phase. Also other backends like the C backend require these changes for garbage collection to work. The semantics of the original program should be preserved by these changes.

Following changes to grin-to-silly phase are done to ensure that garbage collection can work correctly:

- Propagate meta information of function local variable types at compile time. This information is used to decide if a local variable is a stack root as explained in section 4.1.1. The actual emitted silly code does not change.
- All grin nodes are now allocated with at least two fields. This is needed to be able to store forwarding pointers and holes as explained in section 4.1.3.
- All nodes need to have a valid constructor value after allocation and before a garbage cycle could possibly occur. Currently garbage collection could occur only when doing a new heap allocation. For all grin nodes stored on the stack this is already the case, because these values are used directly. But heap space for the RP and globals is prematurely allocated when the application starts, and will be used eventually when the program runs. When a garbage collection is needed, the collector algorithm will scan the contents of globals. If no valid grin tag is stored already, the collector can crash because it expects it can inspect the tag. Solution for this is to temporarily store harmless values that will please the garbage collector.
- Hole allocations now save the max arity in the second field. This is explained in section 4.1.3.

4.2 Related work

4.2.1 Differences with other UHC backends

We will now compare the implementation of LLVM garbage collection we described in this chapter with the implementations of the other UHC backends. The bytecode backend and the C backend do also use the UHC garbage collection library. Our observation is that all three implementations are very similar. But for finding the garbage collection roots on the stack the backends take a different approach.

For finding the garbage collections roots on the stack, the LLVM backend uses the build in shadow-stack. This is explained in chapter 3. The bytecode backend uses static stack descriptors. Extra information is saved to aid the garbage collector. This information is not stored on the stack, but in a separate datatype. For every function a stack map is stored that describes at which offset the garbage collection roots are located. The descriptors use run-length encoding. Run-length encoding is a very simple encoding. A numerical value indicates how many subsequent positions are not stack roots. Then a numerical value describes how many subsequent stack positions are stack roots. This is done for the whole stack, dividing the stack in regions with stack roots and without stack roots. This is all information what is needed for the garbage collector to walks the stack and can find the roots. This approach is possible because there are no optimizations performed on the generated bytecode that can change the layout of the stack after the descriptors have been generated.

4.2.2 The GHC LLVM backend

In this section we discuss the difference between the UHC LLVM backend and the GHC LLVM backend. We take a closer look to the design decisions and implementation choices made to see what differences are with our implementation. We especially focus on the differences concerning garbage collection.

The front-end of the GHC and UHC compilers first compile the Haskell program to an intermediate representation (IR) of Haskell. Both compilers use the same name for this language: Core. But the languages are not equal, but the idea is the same. It is a simplified desugared version of Haskell similar to lambda calculus.

The UHC compiles the Haskell program to grin, but the GHC compiler compiles the program to STG. The goal of grin and STG is to express more concrete implementation details, like the evaluation of lazy evaluation. The languages are similar. Garbage collection does not play a role at this point because memory allocation is not made explicit. So we do not go into more detail.

After STG and grin, UHC compiles the program to silly and GHC to Cmm. This phase is more interesting. Cmm and silly are low level intermediate languages with an imperative look and feel. The style of the languages is similar to the programming language C. Both languages have an unbounded amount of variables, abstracting from real hardware registers. Also both languages support if statements, while constructs and function definitions and functions calls. Silly stands for simple imperative little language. This language has a smaller feature set than Cmm. The Cmm language is a subset of c- -. It contains a well specified datastructure for the stack. The GHC has full control of this piece of memory and the LLVM optimizations will not change the format. Garbage collection is therefor solved at the level of Cmm, and not at the level of LLVM. The extra work needed for garbage collection to work has already been done so the GHC LLVM backend can benefit from it.

We think that using the LLVM stack will produce better LLVM optimization results, resulting in better performance. But on the other hand for this approach the shadow-stack is needed and this also results in a runtime overhead. More research is needed to draw conclusions about the real impact on performance of both approaches.

4.2.3 LHC compiler

The LHC compiler is also a Haskell compiler with an LLVM backend. The compiler also uses the grin intermediate language for optimizations. The LHC does not have an accurate garbage collector. We can not discuss the differences concerning garbage collection. The creator of the LHC compiler states on his blog that the lack of decent support for garbage collection by the LLVM framework keeps him from implementing accurate garbage collection. The overhead of using a shadow-stack will badly influence the performance of functional languages [Him09]. The LHC compiler currently uses the Boehm conservative garbage collection library.

4.3 Observations and conclusions

In this chapter we described how we implemented support for accurate garbage collection for the LLVM backend. We combined the LLVM framework, the UHC compiler and the UHC garbage collection library to make this possible.

- The LLVM shadow-stack provides by the LLVM framework helps us by finding the garbage collection roots on the stack and deals with the LLVM optimisations.
- The ability LLVM provides to tag the roots with metadata is not useful for our implementation.
- The UHC garbage collection library is versatile enough to cooperate with the LLVM backend. Two plug-ins are needed to help the garbage collector finding the roots and tracing the memory.

In next chapter we will test our implementation and compare the results.

Chapter 5

Results

In previous chapter 4 we explained how we implemented accurate garbage collection for the LLVM backend. Now we have implemented it, we want to test it against real world programs to see if everything is functioning correctly. If the benchmark programs are working without problems, we are a bit more sure that everything is correctly implemented.

We also want to know if our expectations are right. In chapter 1 we said that a part of our motivation for replacing conservative with accurate garbage collection is the expected improvement of performance. We will investigate if this is the case by comparing the performance of the old Boehm implementation and the new accurate implementation.

Besides the individual improvement of the LLVM backend, we will also measure how the LLVM backend performs in comparison with other UHC backends, like the C backend and the bytecode backend. This is interesting because we expect that the LLVM backend can help the UHC compiler to generate efficient low level code. Now we have accurate garbage collection for most backends, the benchmarks are less influenced by garbage collection, so we can make a better comparison of generated code efficiency between the different backends. We want to see if the LLVM backend is a promising backend.

But the first thing we want to check is something we came up with along the route. In chapter 2 we wondered what the overhead of the shadow-stack approach will be in practice. We want to measure the runtime overhead of maintaining the shadow-stack to answer the question: does the shadow-stack transformation results in acceptable overhead?

5.1 Nofib suite

The nofib suite is a benchmark suite developed by the Glasgow Haskell compiler group [Par93]. It contains various benchmarks to measure and compare the performance of haskell compilers. The benchmarks contain source code and sample inputs.

5.1.1 Benchmarks set

The nofib suite consists of multiple subsets of benchmarks. There is a subset called real, with large and complex test programs. There is another subset called spectral. Tests in this set only focus on specific properties of programs. For our benchmark we only used the imaginary subset. There is a simple reason for this. The LLVM backend has insufficient support yet to compile the real or spectral subsets. The real subset depends a lot on IO. Adding support for extra features like IO was beyond the scope of this project. So the runtime benchmarks are maybe not a good reflection of how fast the system is. But it is a good first step to get an idea.

	Shadow-stack disabled (sec)	Shadow-stack enabled (sec)	Ratio
digits-of-e1	9.22	12.44	1.35
digits-of-e2	6.80	8.23	1.21
exp3_8	2.01	2.80	1.39
primes	1.28	2.24	1.75
queens	2.00	3.20	1.60
tak	1.22	1.98	1.62
wheel-sieve-1	9.33	41.70	4.47
wheel-sieve-2	0.72	1.73	2.40
average			1.62

Table 5.1: Shadow-stack overhead

5.1.2 Test system configuration

The nofib suit advises that the results must be published accompanied with the test setup. The tests are ran on a Intel core2duo processor with 2.4 Ghz. The system has 1 gigabyte of main memory. The test are compiled and ran on Ubuntu 9.10 operating system (32 bit). We used the UHC release 1.1.0 in combination with LLVM framework version 2.6. All tests are executed with the same workload.

5.2 Shadow-stack overhead

We want to make a fair measurement of the shadow-stack overhead. We will measure the real wall clock runtime of the benchmarks.
Two situations will be compared:

- Shadow-stack enabled. A normal run of the new implementation including the code to maintain the shadow-stack.
- Shadow-stack disabled. All machinery to dynamically maintain the shadow-stack is removed from the code.

To be able to make a fair comparison, we have to take care of one extra thing. When the shadow-stack is disabled, we can not perform garbage collection for obvious reason. This is not a problem for still being able to create functioning executables. We can let the programs only allocate memory and never deallocate the allocated memory. We achieved this by replacing the memory allocation calls to the garbage collection library with the build in malloc from the LLVM instruction set. This will result in programs that still work but will claim a lot of memory. If we do this for both benchmarks the comparison will be fair, because time needed for garbage collection is kept of of the picture.

5.2.1 Observations

The results of the shadow-stack benchmark are shown in table 5.1. The first column shows the results of the tests with the shadow-stack disabled. The second column shows the results of the test with the shadow-stack enabled. The third column shows the ratio of two measurements. We can see that the runtime indeed suffers from maintaining the shadow-stack. All tests results with the shadow-stack enabled are slower. The ratios range between 1.21 and 4.47 times. We must note that the test with the worst result, the wheel-sieve-1 benchmark, does not have a representable result. This benchmark used a lot of memory, and with the shadow-stack enabled the system ran out of memory and started using the swap memory on disk. Programs that start using swap space are always much slower. So it is a good example of how using more memory can trigger a bottleneck. But we consider it noise and did not include this measurement in the calculation of the average ratio. We calculated that on average the runtime is 1.62 times slower with the shadow-stack enabled. Now we need to do some more test to see if this is an acceptable overhead.

5.3 Comparison of conservative and accurate garbage collection

This set of benchmarks compares two situations:

- Conservative garbage collection. We used an old version of UHC for this test. This old version has an LLVM backend that makes use of the conservative Boehm garbage collection library.
- Accurate garbage collection. For this test the new UHC release is used, combined with the new LLVM backend that includes accurate garbage collection.

The results of this test can be found in table 5.2. Two tests did not work correctly, indicated by crosses in the table. The Boehm version of digits-of-e1 did not compile. The old backend we used for the Boehm tests contained a compiler bug. We think it is not worth the effort to fix this bug because the old backend will be replaced by the new accurate implementation. We still produced enough useful results.

The second test that failed is the queens benchmark for accurate garbage collection. This test should work with the new implementation. The queens test does compile, but produces a segmentation fault when it is executed. We investigated this problem and we found the bug that is responsible for this segmentation fault. The bug is caused by a fundamental problem in the grin to silly translation. Some bad values will sometimes be copied from the RP to positions that are registered as garbage collection roots. The mutator will not crash because it will never use them. But when the garbage collector encounters these bad values, it will crash. It is an exceptional situation, and the C backend might suffer from the same problem. A solution for this bug is not yet implemented, but it will be solved in the future. Testing the new implementation with the benchmarks brought up this bug so testing is shown useful for this reason.

5.3.1 Observations

The first observation we like to discuss is observed when comparing the accurate garbage collection implementation results with the results of the benchmarks in previous section. In previous test we measured the overhead of the shadow-stack. All test in this chapter are executed with the same workload so we can compare them with each other. The benchmarks in previous section do not perform garbage collection, but only claim memory with the build in LLVM malloc instruction as explained. We can compare the accurate garbage collection results in table 5.2 with the shadow-stack enabled results from table 5.1, because the accurate test also has the shadow-stack enabled. We can see that the benchmark results with accurate garbage collection are better than the benchmark results without garbage collection. This might be caused by the fact that the build in malloc allocator from the LLVM framework is less inefficient than the allocator of the UHC garbage collection library. The UHC allocator is fast because it is combined with a two space copying garbage collector. The advantage of this combination is explained in section 2.2.5.

If we compare the results of accurate garbage collection with the results of conservative garbage collection we can see that it is always faster. The results are shown in table 5.2. The performance gains range between 1.39 and 4.97 times faster execution. The average performance gain is exactly 3.00 times. This is a very nice result, especially if we consider that this includes the shadow-stack overhead. John van Schie expected that a large part of the execution time was taken by garbage collection [VS08]. We can now verify that this claim is correct.

5.4 Comparison of different backends

We now compare the results of three different backends. We compare the LLVM backend with the C backend, and the bytecode backend.

	Runtime (sec)		
Benchmark	Conservative	Accurate	Ratio
digits-of-e1	x	6.84	x
digits-of-e2	6.40	4.61	1.39
exp3_8	2.10	0.47	4.46
primes	1.68	1.02	1.65
queens	2.50	x	x
tak	1.54	0.31	4.97
wheel-sieve-1	38.12	10.58	3.60
wheel-sieve-1	1.56	0.82	1.90
average			3.00

Table 5.2: Conservative versus accurate garbage collection

5.4.1 Comparison with the bytecode backend

When we look at the results at table 5.3 we can see that the LLVM backend is much faster than the bytecode backend. The differences range between 2.73 times faster and 5.05 times faster execution times for the LLVM backend. The average advantage is 3.79 times faster compared with the LLVM backend.

This is what we expected because the bytecode backend has a different focus. The bytecode backend focuses on fast compilation times, and not on producing a fast executable. The LLVM backend delivers a faster executable because it benefits from whole program analysis and many optimizations. But we included the results to give an idea of the difference.

5.4.2 Comparison with the C backend

The comparison between the LLVM backend and the C backend is more interesting. The LLVM and C backend share a large part of the compilation pipeline and optimizations. Programs compiler with the LLVM and C backend will also use more or less the same amount of memory. The only differences is how the machine specific low level code is generated. The machine instructions for the C backend are generated by the C compiler GCC. The machine instructions for LLVM will be generated by the LLVM compiler. We want to see if the LLVM framework can do a better job in producing machine specific code. We can now have a fair comparison because both the LLVM an C backend have accurate garbage collection. The focus can now be more on the performance of the generated code for the mutator because the share of garbage collection is become less.

When we look at the benchmark results in table 5.3 we can see that the C backend is faster for only one benchmark. The digits-of-e1 benchmark is slightly faster for the C backend. The LLVM backend is faster for all other benchmarks with results ranging between 1.11 and 2.06 times faster execution. The average is 1.34 times faster execution time for the LLVM backend.

John van Schie measured the performance difference between the C and the LLVM backend in his thesis [VS08]. He measured an average of 1.10 times advantage for the LLVM backend. At that time the C and LLVM backend used conservative garbage collection. Now with the accurate garbage collection implemented for both backends we measured a performance of 1.34. The difference between the two backends has become larger because of accurate garbage collection. This is expected because the share of garbage collection is decreased, and the code for the mutator remains the same.

John van Schie did a guess of the impact of accurate garbage collection. He based his guess on the results with garbage collection disabled. The same approach was used by replacing the garbage collection with allocation without deallocation. We did the same for measuring the impact of the shadow-stack. John expected that with

Benchmark	LLVM	C	C ratio	Bytecode	Bytecode ratio
digits-of-e1	6.84	6.69	0.98	21.00	3.07
digits-of-e2	4.61	5.46	1.18	13.84	3.00
exp3_8	0.47	0.56	1.19	2.26	4.81
primes	1.02	1.36	1.33	3.03	2.97
queens	x	0.74	x	3.11	x
tak	0.31	0.64	2.06	1.51	4.87
wheel-sieve-1	10.58	11.77	1.11	53.42	5.05
wheel-sieve-1	0.82	1.23	1.50	2.24	2.73
average			1.34		3.79

Table 5.3: Backends comparison

accurate garbage collection the result could be 1.17 in stead of 1.10. We can now say that the results is even better. We measured the average result of 1.34. We think that we have an explanation for the even better result. John based his guess on the measurements using LLVM malloc. We observed in previous section 5.3.1 that the LLVM malloc is inefficient. So now with an copying garbage collector with an efficient allocator the results are even better than expected.

Chapter 6

Further work

During this research project we successfully implemented accurate garbage collection for the LLVM backend with good results. But there is always room for improvement. Also the UHC project is under active development and features are added and changes are made frequently.

We divide this chapter in two parts. First we like to discuss what improvements could be done and additional features could be added in the future concerning garbage collection. Then we will discuss what impact some expected changes to the UHC compiler will have on the current garbage collection implementation.

6.1 Garbage collector

We will first take a look at some possible improvements for the garbage collector.

6.1.1 Tagless garbage collection

The garbage collector needs to know how to collect and trace all in memory objects. Currently memory objects are annotated with a tag field. The garbage collector uses this field to lookup meta information about the memory object. This meta information describes what kind of data the fields of the node contain. The meta information includes for example object size, and what values are pointers so they can be followed.

Looking up this information, dynamically inspect the object and make decisions about how to collect the node results in runtime overhead. Every memory object is accessed and dynamically inspected during a collection cycle. When running a large program, there are very many objects in memory. The overhead per object might be small, but the code is executed many times. So this piece of code is an important candidate for optimization.

Another approach could be to generate a specialized collection function for every node [Gol91]. This function performs all necessary actions directly for the particular node. The compiler inspects all different types of nodes and generates a specialized function for every node at compile time. But how do we know which function we need to call for a specific node when performing garbage collection? In stead of storing a tag for every node, we could also store the function pointer for the specialized function. Then we only have to jump to this code and it will perform the actions for the collection. We eliminate the cost for runtime node inspection by trading in for compile time inspection resulting in a performance gain. Disadvantages are that the compile time will become longer, and the executable will become larger, depending on the amount of specialized functions needed.

Generating a specialized function for every possible node will result in an enormous amount of extra functions needed only for garbage collection. The amount of needed specialized functions could be reduced by creating specialized function for similar object types. For example a functions that will work for all nodes that has two fields and only contains pointers. This can prevent an explosion of code size generated by the compiler.

Implementing tagless garbage collection has impact on the compiler. There are two aspects we need to take care of:

- The garbage collector is not the only piece of code making use of the node tags. Also during the evaluation of nodes the tags are used to identify the object. Currently a simple unique integer value is used to identify nodes. The only criteria for the tag is that it should be unique for every node type. Function pointers could also be unique numbers, so changing this should not cause problems.
- The same idea used for specialized functions for garbage collection, could be utilized for others things as well. The inspection of the tag for evaluating functions could also be replaced by a specialized evaluation function. We can only do this trick once, or we have to store multiple specialization function pointer per node. The multiple specialization pointers are the same for every unique node. So we can do even better by using only one pointer that points to a set of specialization pointers that are collected for every node and stored separately. Then we do not have to store redundant information.

6.1.2 Stack descriptors

An important task of the garbage collector is to find all the garbage collection roots. In the implementation chapter 4.1.1 we explained how we find the garbage collection roots using the shadow-stack. The shadow-stack is directly supported from the LLVM framework and is simple to use. The shadow-stack method has also some disadvantages as explain in section 3.3.3. The most important disadvantage is the runtime overhead. This runtime overhead is measured in section 5.2. There might be a more efficient method with less overhead.

The LLVM Framework has support for generating stack maps. Metadata about the gcroots is collected when generating target code. This meta data might be useful to add extra garbage collection specific code during compile time. This requires extending the LLVM compiler by implementing a compiler plugin. To be able to implement this detailed knowledge about the LLVM framework is needed. Implementing more specialized garbage collection code for stack root finding might result in better performance.

Also the LLVM framework can help us at this point with a new release. The LLVM framework can come with a better compiler plug-in that generates faster code. Then we only need to select another compiler plug-in. We use the garbage collection interface LLVM recommendeds. If the LLVM framework comes with a more efficient implementation for finding stack root, our garbage collector might work out of the box. But we expect that the datatypes or design will slightly change so the only thing we need to do is to rewrite the stack walking plugin. This does not affect other backends.

6.1.3 Generational garbage collection

The collector of the UHC garbage collection library uses a two space copying algorithm. In the future this algorithm might be replaced with a generational garbage collector. Generational garbage collection is based on the generational hypothesis. This hypothesis states that most recently created objects are likely to become unreachable quickly. Multiple hierarchical memory regions are defined and objects that survive a specific collections are promoted to a higher region. Memory regions are called generations. Generations are collected independently and higher generations are collected less frequently because because the objects in there are expected to have a longer lifetime. Higher generations are often called older generations because they tend to contain older objects. Generational garbage collection is a heuristic approach that offers good performance in practice.

Performing a collection on one specific region at a time is has one problem. We must be able to find all pointers from older generations to objects in the collected region. Older generations may have pointers to younger generations, however this is not occurring very often. Otherwise we can not be sure a object has become garbage. We also do not want to trace all objects in all generations because then we loose the benefit of collection only one generation at a time. To solve this problem a technique called remembered sets [Ung84] is used. A write barrier checks all writes to objects to see if pointers from older to younger generations are stored.

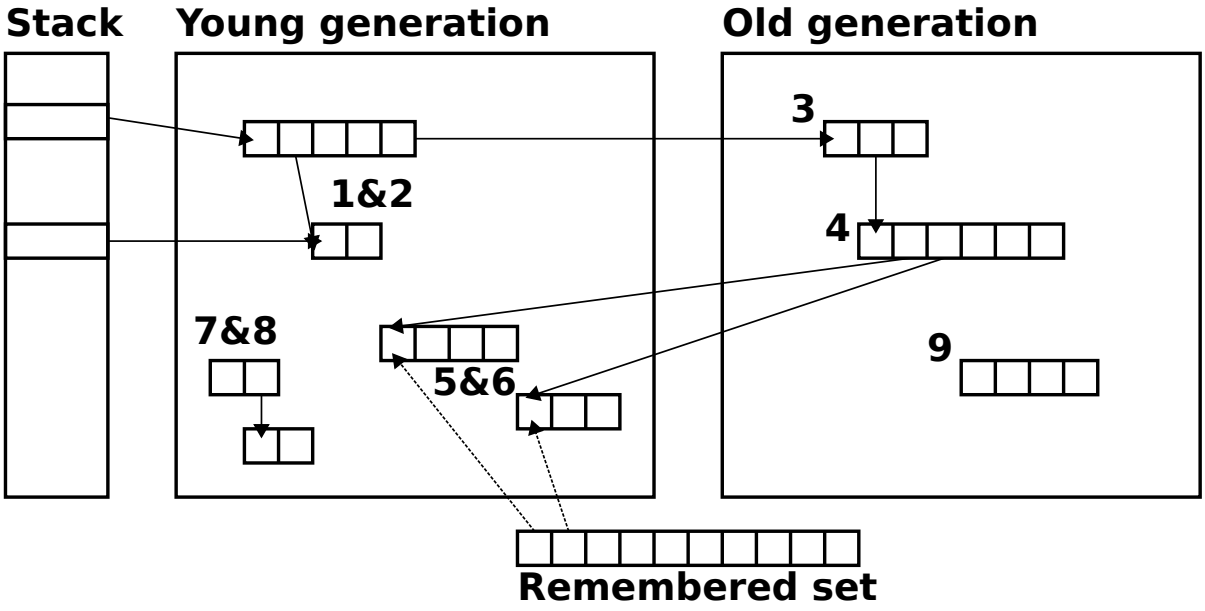


Figure 6.1: Remembered sets

Theses writes are recorded in a remembered set and can be treated as a set of extra roots ensuring that some objects are still live.

An example of generational garbage collection with remembered sets is shown in figure 6.1. Suppose we want to collected the young generation. Node 1 and 2 live because they are reachable from the stack. Node 3 and 4 are also live because they are reachable through node 1, but we do not take them into account when collection because we only want to collect the young generation. Because we do so, node 5 and 6 might be considered garbage because there are no pointers to 5 and 6 within the young generation. But because we administrated all pointers from the old generation in the remembered set, we know that 5 and 6 are also live. Node 7 and 8 are garbage and will be collected this cycle. Note that node 9 is also garbage but only collected when collecting the old generation.

The impact of implementing generational garbage collection is that we need to be able to define write barriers. The LLVM Framework has support for write barriers as explained in section 3.4. This makes it easy to insert special code to perform extra actions when writing to specific memory addresses. We belive that without the support from the LLVM Framework it would also be possible to insert extra write barrier code. The write barrier support is more like a nice construction to separate concerns.

6.2 UHC changes

The accurate garbage collection implementation must be prepared for design and implementation changes of the UHC compiler. Some additions to the compiler might break the llvm garbage collector. We investigate some features that might be implemented in the future and describe the impact on the llvm garbage collector.

6.2.1 Unboxed datatypes

We explained in chapter 4 how the grin nodes are stored in memory. Important observation was that we could distinguish between primitive and non-primitive nodes. Non primitive nodes have zero or multiple pointer fields, and primitive nodes always have one non pointer value. This is a very basic representation and there are optimizations possible. For example we could store primitive values like integers directly in nodes. The difference between boxed and unboxed nodes are visible in picture 6.2. Unboxed nodes have advantages. First we need less heap space to store the pointer and the tag. Also we have less indirection. This will result in faster memory access for the mutator. But also the garbage collection can be performed faster because less nodes need to be inspected.

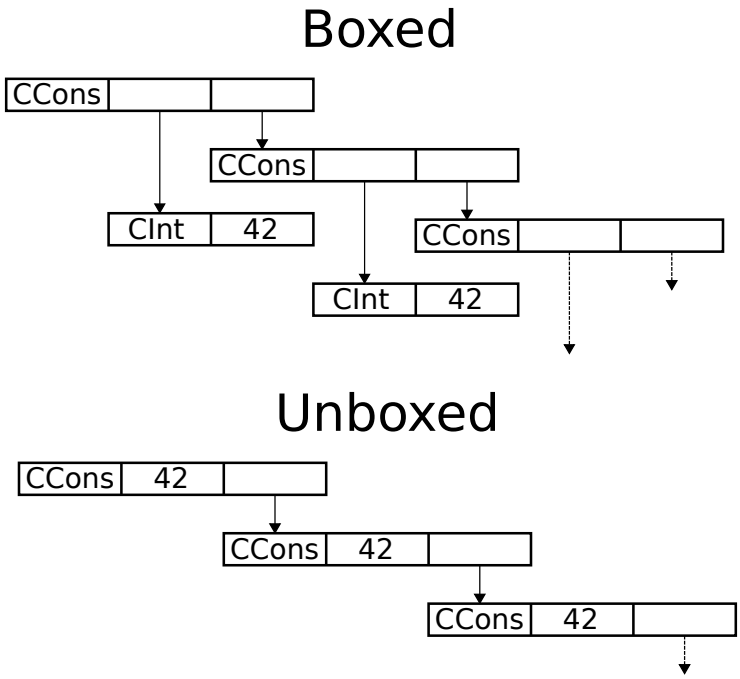


Figure 6.2: Boxed versus Unboxed primitives

Having support for unboxed datatypes has impact on the compiler and also the garbage collector. The grin node descriptors we introduced in the implementation chapter are not sufficient anymore. We can not only make a difference between primitive or non-primitive, we have to provide a complete description of the node describing every field of the node. The main focus here is on which fields are pointers. We need to be able to find all pointers to be able to trace all live nodes. When we have extended the descriptors we also need to change the collection algorithm to make use of this new information.

6.2.2 LLVM support for higher variant

The UHC compiler is actually a series of compilers. The code is divided an categorized in variants. The lowest variant only includes basic functionality. Every variant adds one or more features. So higher variants include more features. The UHC compiler is build for a specific variant.

Current LLVM backend has support for all variants up to variant 8. Other UHC backends have evolved to support up to variant 99. So various other steps of the UHC compiler have support up to variant 99. The LLVM backend could also support variant 99, but for this to work the LLVM must be able to handle all new features

introduced by variants higher then 8. New functionality provided by higher variants include: Type classes, IO, Exceptions, Standard Library support.

Exceptions are not implemented for the whole program analyze pipeline so we do not need know what impact this will have. Most other features like Type classes and the standard library do not require extra support from the LLVM backend. Code that makes use of these features will automatically work because it will generate Silly code the Silly to LLVM transformation is already able to handle. The only relevant feature is IO. For IO special grin nodes are used. IO grin nodes typically store pointers. The pointers are used for files and streams. Previously pointers where only used for heap locations. So we have to differentiate between IO pointers and heap pointers. The garbage collector only needs to handle heap pointers.

Chapter 7

Conclusions

In this chapter we briefly summarize the experiences we gained during the research project.

7.1 Implementation

When we started this research project we asked ourself a few questions. At first we wanted to know what impact adding support for accurate garbage collection would have on the LLVM backend. We found out that the UHC compiler must pass through some extra information. This information is used to create a description table that the garbage collector could use to perform reachability analysis. Also some changes had to be made to the pipeline and order of generated code.

The LLVM framework provides support for implementing garbage collection. Unfortunately, the support is very primitive. The LLVM does not implement a garbage collection algorithm. But it provides a compiler plugin that can maintain a dynamic structure on the heap called a shadow-stack. The shadow-stack can help us to find garbage collection roots on the stack. Creating a stack map was one of the largest problem we expected and the LLVM framework helped us nicely with this. The shadow-stack was a good solution to deal with the aggressive optimizations the LLVM compiler performs.

The changed made to the LLVM backend could be used to create garbage collector plug-ins for the UHC garbage collection library. The UHC garbage collection library seemed versatile enough to cooperate with the LLVM backend by only creating plug-ins and no other changes where needed to the library itself.

7.2 LLVM framework

We believe that the LLVM framework could support garbage collection even better by supplying an generic garbage collection algorithm implementation like the UHC garbage collection library. The main argument of LLVM to not provide an algorithm implementation is that it would never we versatile enough. In contrast the UHC garbage collection does a good job at this point. We believe that it would even be possible to create a garbage collector for an object oriented language. In this case we would not use the LLVM garbage collector library because the UHC has its own library, but there is some room for improvement here for the LLVM framework. Other compiler writers can benefit from a garbage collection algorithm implementation when it is included in the framework.

7.3 Performance

After the accurate garbage collection was implemented successful we measured the performance of the new implementation. Our motivation for this research project was that accurate garbage collection could give compiled applications a large performance improvement. This was indeed the case. On average the accurate garbage collection implementation is 3.00 times faster than the conservative implementation. This includes the overhead of the shadow-stack. We could say that the overhead of the shadow-stack is acceptable. The shadow-stack makes the overall performance gain possible. John van Schie claimed that conservative garbage collection was responsible for a large part of the execution time [VS08]. We can now support this and say that the influence is even bigger than expected.

7.4 Future

The benchmark results showed that the LLVM backend is the fastest UHC backend. The performance difference is increased since garbage collection takes less share in the overall execution time. The LLVM backend is a promising backend. It can generate efficient low level code for many hardware platforms. More work should be spend on making the LLVM backend fully functional, because currently the functionality is a bit behind on other backends.

Bibliography

[Boq99] U. Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, April 1999.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[DFS09] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The Architecture of the Utrecht Haskell Compiler. In *Haskell Symposium*, 2009.

[ECM06] ECMA. Standard ecma-335: Common language infrastructure (cli). http://www.ecma-international.org/publications/standards/Ecma-335.htm, 2006. [Online; accessed 25-August-2010].

[Gol91] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 165–176, New York, NY, USA, 1991. ACM.

[Hen02] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *In Proceedings of the Third International Symposium on Memory Management*, pages 150–156. ACM Press, 2002.

[Him09] David Himmelstrup. Why llvm probably won't replace c–. http://lhc-compiler.blogspot.com/2009/01/why-llvm-probably-wont-replace-c.html, 2009. [Online; accessed 25-August-2010].

[JL07] Richard Jones and Rafael Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons Ltd, 2007.

[LA04] 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.

[Par93] Will Partain. The nofib benchmark suite of haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGPLAN Not.*, 19(5):157–167, 1984.

[VS08] John Van Schie. Compiling Haskell To LLVM. Master's thesis, 2008.