TU WIEN Informatics

SSA transformation for GHC's native code generator

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Benjamin Maurer
Matrikelnummer 00826765

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 1. März 2021

Benjamin Maurer   Andreas Krall

Informatics

SSA transformation for GHC's native code generator

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Benjamin Maurer
Registration Number 00826765

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, 1st March, 2021

Benjamin Maurer	Andreas Krall

Erklärung zur Verfassung der Arbeit

Benjamin Maurer

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. März 2021

Benjamin Maurer

Abstract

Static Single Assignment (SSA) form is a widely-used and effective intermediate representation in optimizing compilers, which explicitly encodes def-use chains and enables a wide range of optimizations. However, transforming native machine instructions to SSA poses additional challenges, due to register constraints, instructions modifying input operands and other Instruction Set Architecture peculiarities. The aim of this practical work is to implement SSA construction and destruction in the Native Code Generator backend (NCG) of the Glasgow Haskell Compiler (GHC). Beyond enabling future optimizations, this will discover and rename webs for improved register allocation. This leads to a reduction in inserted spill instructions of up to 27% for the graph coloring register allocator.

Contents

CHAPTER 1

Introduction

1.1 Haskell and GHC

Haskell[1] is a purely functional programming language with lazy evaluation semantics. To achieve pure functions, Haskell has pioneered the use of monads to model side-effects[JW93]. It features a rich, static type system, which can be used to model program constraints at compile time.

The main compiler for the language - the Glasgow Haskell Compiler (GHC) - was first released in 1992 (see [HHJW07]) and differs in its internal architecture from more mainstream compilers for imperative languages.

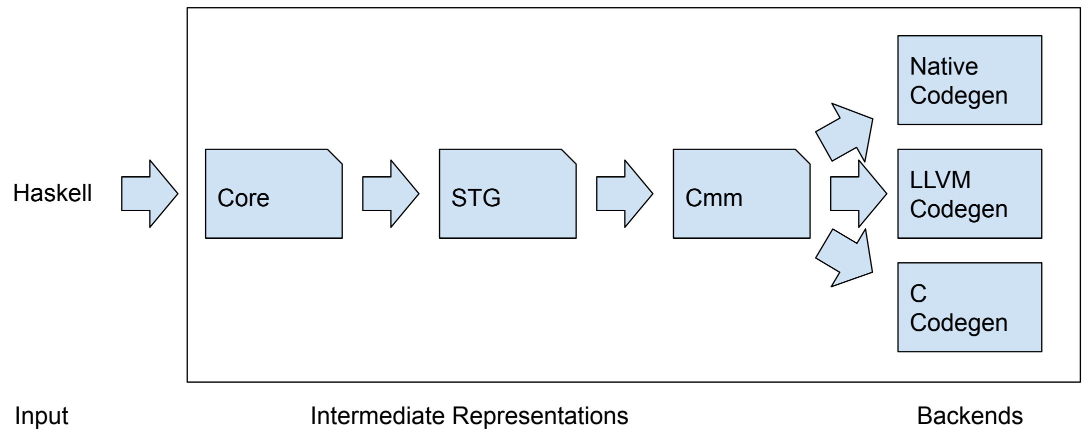

Figure 1: GHC compilation pipeline

[1]https://www.haskell.org/

Figure 1 gives a high-level overview of used intermediate representations in GHC. After parsing, GHC translates the program into an intermediate representation (IR) called "Core", a simple, typed functional language based on $SystemF_C$ [SCJD07]. Many optimizations are performed in a "Core to Core" pass. Next, Core is lowered to STG, a minimalist functional language for an abstract machine called the "Spineless Tagless G-machine" [Jon92], which aims to facilitate mapping of a non-strict functional language to actual hardware. STG is then lowered to Cmm, GHC's implementation of the C-- language [JRR99], a low level imperative language with an explicit stack.

GHC currently offers three backends. The C-backend[2] is the oldest one and considered deprecated except for bootstrapping on new platforms. A backend utilizing the LLVM[3] compiler infrastructure ([TC10]) is the most recent of the three, but suffers from prohibitively long compile times. GHC's default backend is the Native Code Generator backend (NCG), which strives to balance compile times and speed of generated code. NCG is relatively simple and performs instruction selection, CFG-based basic block layout optimization, liveness analysis with dead code elimination and register allocation. The vast majority of optimizations is performed in earlier phases.

At this stage, no intermediate representation is used, but a stream of machine instructions with either physical registers or virtual registers (before register allocation) and some meta-instructions (e.g., `SPILL`, `RELOAD`), which are substituted in the course of compilation. Two register allocators are available, the default linear scan allocator and a Chaitin-Briggs-style graph coloring allocator.

1.2 Motivation

While Haskell is known to be at the cutting edge in terms of type system features, like adding generalized algebraic datatypes (GADTs), type families, kind polymorphism and recently linear types [BBN$^+$18], code generation has received less attention.

The graph coloring register allocator has been removed from the standard optimization flags (-O2) in 2013[4], as it suffered a regression in performance of generated code and the situation has not been remedied since.

Analyzing the short routine in Listing 1 submitted to GHC's issue tracker[5], which exhibits excessive spilling, reveals one fundamental problem.

The routine simply adds three records with nine `Int` fields each together in some arbitrary permutation. Note the use of the `BangPatterns` language extension[6], which forces

[2]`https://downloads.haskell.org/~ghc/9.0.1/docs/html/users_guide/codegens.html#c-code-generator-fvia-c` – Accessed 5.3.2021

[3]`https://www.llvm.org/` – Accessed 5.3.2021

[4]`https://gitlab.haskell.org/ghc/ghc/-/issues/7679` – Accessed 5.3.2021

[5]`https://gitlab.haskell.org/ghc/ghc/-/issues/8048` – Accessed 5.3.2021

[6]`https://downloads.haskell.org/~ghc/9.0.1/docs/html/users_guide/exts/strict.html` – Accessed 5.3.2021

```
{-# LANGUAGE BangPatterns #-}

module Spill where

import GHC.Exts

data S = S !Int !Int !Int !Int !Int !Int !Int !Int !Int
         deriving Show

spill :: S -> S -> S -> S
spill (S !a !b !c !d !e !f !g !h !i)
      (S !j !k !l !m !n !o !p !q !r)
      (S !s !t !u !v !w !x !y !z _)
 = S (a + j + s) (b + c) (k + r)
     (a + b + c + d + e + f + g + h + i)
     (j + k + l + m + n + o + p + q + r)
     (s + t + u + v + w + x + y + z)
     (a + b + c) (j + k + l) (s + t + u)
```

Listing 1: Example program triggering spill code insertion

strict evaluation of matched patterns. When compiled using the graph coloring register allocator[7], it will insert 8 store and 9 load instructions for spilling.

Inspecting one spilled virtual register (vreg), `%vI_nKI`, we find that it is live in two basic blocks, as can bee seen in Figure 2.

Looking at the assembly in Listing 2, we can see that vreg `%vI_nKI` gets redefined several times, explicitly stored on the stack between the two basic blocks and even dies and gets redefined within the same block. Using the same virtual register name constrains the graph coloring allocator to assign these disjoint live ranges to the same physical register, increasing register pressure. These longer live ranges also may have more conflicts with other live ranges, increasing likelihood of spilling.

1.3 Renumbering

Generally, a Chaitin-Briggs-style allocator would have a *renumbering* phase, which discovers intersecting def-use chains (webs) and assigns them unique virtual register names [BCT94].

This is both important before register allocation can begin, in order to identify life ranges, as well as during register allocation to identify newly created live ranges (e.g., via

[7]Using GHC 8.10.3, with `ghc -O2 -fregs-graph -ddump-asm-stats`

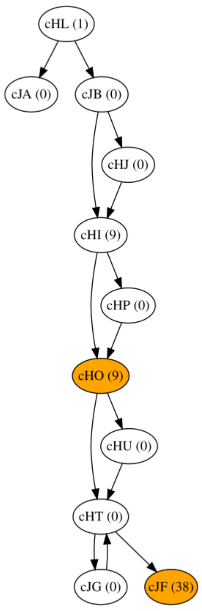

Figure 2: Control-flow graph of Listing 1, nodes with virtual register occurrence highlighted

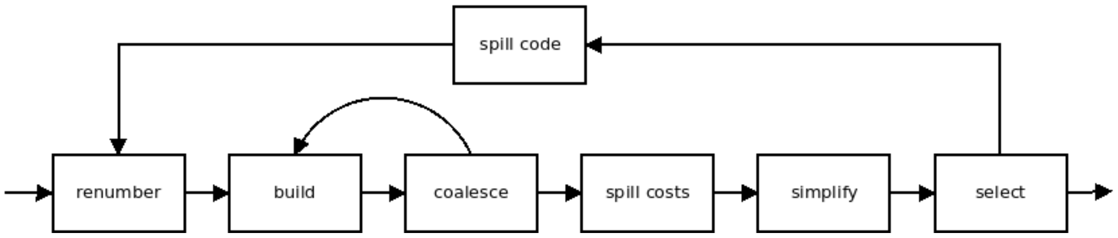

Figure 3: Phases of a Chaitin-Briggs-style allocator

spilling or live range splitting). No such phase exists in NCG. vreg's are created during instruction selection in the Cmm-to-ASM pass. Further optimizations may disjoint live ranges without renaming them. NCG's graph coloring allocator only renames live ranges during spilling, where it can locally generate a new name as it emits `SPILL` and `RELOAD` meta-instructions around the original instruction.

One approach to *renumbering* uses classical iterative data-flow analysis. Def-use chains are discovered by performing *reaching definitions* analysis, then pair-wise intersection tests are performed to find common uses and union these intersecting def-use chains together into a web. Representing def-use chains as lists, not only requires $m * n$ space, for m definitions and n uses, but is also computationally expensive.

Briggs' seminal paper on *optimistic coloring* [BCT94] mentions that Chaitin's original graph coloring register allocator [CAC+81] used this approach, whereas they utilized Static Single Assignment (SSA) form.

```
cHO
    movq $block_cHT_info,-64(%rbp)
    movq 7(%rbx),%vI_nKI   # Definition of nKI
    # [...]
    movq %vI_nKI,80(%rbp) # nKI dies
    addq $-64,%rbp
    testb $7,%bl
    jne _blk_cHT
    jmp _blk_cHU

cJF
    # [...]
    movq 144(%rbp),%vI_nKI # Definition of nKI
    movq %vI_nKI,%vI_nJY    # nKI dies
    # [...]
    movq %vI_nKC,%vI_nKi    # Definition of nKI
    # [...]
```

Listing 2: Excerpt of generated assembly of spilling example

Transforming the program into SSA form will create a new name for each definition of a value and webs can then be simply found by applying the union-find algorithm. The big advantage though of SSA form is, that it enables a wide range of further optimizations.

1.4 Static Single Assignment Form

SSA form is widely used in modern compilers, as it facilitates many optimizations and analysis in an efficient manner [Sin18]. Both data-flow and control-flow is explicit in SSA.

When transforming a program into SSA, each (re-)definition of a variable introduces a new name. Generally, an index is added to to the original variable name, incremented with each definition. For example, the following side-by-side comparison before and after renaming:

```
x := 1                  x_0 := 1
x := x + 2              x_1 := x_0 + 2
y := x + 3              y_0 := x_1 + 3
```

As such, each name is defined exactly once and each definition dominates all of its uses.

At points where control-flow merges with conflicting definitions of a variable, ϕ-functions (also known as ϕ-nodes) are inserted. A ϕ-function is a function of the form $x_n := \phi(x_0, x_1, .., x_{n-1})$, where x_n is a new name for the value and ϕ has as many arguments as the basic block has predecessors. Conceptually, a ϕ-function will "select" the right version of x, depending on the actual path taken. ϕ-functions represent parallel copies, such that all ϕ-functions at the beginning of a block are "executed" in parallel and their "results" are in $Live_{In}$ for the respective basic block [SJGS99].

Several different flavors of SSA exist, namely:

- **Minimal SSA**, described in [CFR+91], inserts the minimal number of ϕ-functions for each join-point of the CFG where more than one SSA-name per original name merges.
- **Pruned SSA**, which performs liveness checks to only insert ϕ-functions for variables that are live in a given block. This can drastically reduce the number of ϕ-functions, but comes at the cost of having to perform global data-flow analysis for liveness, plus the liveness checks on insertion [CCF91].
- **Semi-pruned SSA** will insert fewer (dead) ϕ-functions, by performing a linear scan over the code to weed out all local live ranges, that do not cross basic block boundaries [BCHS98].

1.4.1 SSA in the code generator

Some issues arise when choosing a SSA form code representation in a code generator. Machine instructions of a real-world Instruction Set Architecture (ISA) will often not conform to SSA constraints. For example, such an instruction may modify its operands directly, making it impossible to assign a new name to the new value, e.g.: `inc %rax` - which is equivalent to $x_1 := x_0 + 1$, or `add $4, %rax` which is $x_1 := 4 + x_0$. Instructions may also have implicit operands, like status registers, or some ISAs feature conditional instructions, like this example of ARM assembly: `cmp r0, #3; addlt r0, r1, r2` or `if (r0 < 3) then r0 := r1 + r2` in pseudo-code.

Different approaches to handle these challenges will be addressed In chapters 2 and 3.

CHAPTER 2

Related Work

First we are looking at some older papers for approaches to live range discovery without SSA. Chow and Hennessy [CH90] describe using data-flow analysis [Hec77] to solve separately for the live and reaching definition attributes. Disjoint live ranges are not renamed though, as they rely on their later splitting phase. Chaitin et al. [CAC+81] call this step "getting the right number of names" and use global data-flow analysis to achieve this. In his thesis, Briggs [Bri14] describes in more detail how they use union-find on pruned SSA form to this end, noting the efficiency gained having to only union the definitions reaching a ϕ-node, instead of having to union all definitions reaching each use.

[CFR+91], the seminal paper on SSA, describes properties and an algorithm to compute minimal SSA. [CCF91] introduces pruned SSA, which only inserts live ϕ-functions and Briggs et al. [BCHS98] describe some previously overlooked problems in SSA destruction, as well as semi-pruned SSA, which is cheaper to compute, yet almost as small. Braun et al. [BBH+13] present a simple and efficient lazy backwards algorithm to construction SSA. Sreedhar et al. [SJGS99] describe an approach for SSA destruction, [BDR+09] suggest a more efficient algorithm and notes on correctness issues for SSA destruction.

Leung et al. [LG99] discuss the problems regarding naming constraints for SSA form on machine instructions and represent a scheme to convert between SSA and native code. They propose a three phase approach, which first collects all naming constraints, then marks nodes where these constraints are violated and finally a reconstruct phase to repair the code, which is based on Briggs's et al. SSA destruction in [BCHS98].

Dinechin [dD14] looks at the challenges SSA form in the code generator poses, like ISA and ABI naming constraints, non-kill target operands and conditional execution in general. They discuss different representations of instruction semantics and which compilation phases benefit from SSA, i.e., phases before pre-pass scheduling, remarking that there is still debate on the benefits for register allocation. The paper also features an extensive review on techniques for if-conversion, an optimization technique to replace control flow with straight-line code of predicated or conditional instructions.

In [HGG06], Hack et al. present a graph coloring register allocator on SSA-form. They show that the interference graphs of programs in SSA form are *chordal* and that an optimal coloring for such graphs can be found in polynomial time. Furthermore, spilling, coloring and coalescing can be decoupled in this framework. However, coalescing becomes $\mathcal{NP}$-complete.

Linear Scan register allocation has also been extended to work on SSA-form by Wimmer et al [WF10]. Their work uses properties of SSA-form and a block order, in which earlier blocks always dominate later blocks, to significantly simplify the construction of lifetime intervals. Most of the expensive lifetime interval interferences checks can be omitted with the guarantees provided by SSA-form, however the authors note that already split life intervals are no longer in SSA-form and require these checks. The final *resolution* phase also performs SSA destruction. As the core register allocation algorithm stays the same, the quality of generated code remained the same. The authors reported a simplification of the compiler code and measured reduction both in compile times and memory use.

Many code generation and optimization steps, like SSA-destruction, may introduce many register-to-register copies. The task of register coalescing is to eliminate as many unproductive copies as possible and is typically performed as part of register allocation. While it is generally beneficial to remove extraneous copies, it may have a big impact on colorability of the graph, as two coalesced live ranges may have more conflicts. Park and Moon [PM04] provide an overview of coalescing algorithms, as they present their *optimistic coalescing. Aggressive coalescing*, as used in Chaitin's [CAC+81] allocator, will coalesce any non-interfering, copy-related nodes in the interference graph. While this removes many unproductive copies, it may also worsen colorability. Park and Moon note however, that it may also *positively* impact colorability, if both coalesced nodes a and b had an interference with node n prior to coalescing, as this lowers the degree of n by one. *Conservative coalescing*, by Briggs et al. [BCT94], only coalesces two nodes if this is guaranteed not to worsen colorability. Since *conservative coalescing* potentially misses many opportunities for safe coalescing, George and Appel's *iterative coalescing* [GA96] interleaves conservative coalescing with the simplification phase of coloring, therefore creating repeated opportunities for coalescing. *Optimistic coalescing* performs *aggressive coalescing*, but splits live ranges again, when they have to be spilled.

Many improved techniques for SSA destruction also seek to coalesce, or avoid inserting, copies. Dinechin [dD14] gives an overview of SSA destruction algorithms in section 4, discussing their different coalescing approaches.

CHAPTER 3

Methodology

In the first implementation phase, SSA construction and destruction will be performed back to back, in order to discover webs for register allocation. Because of this, we compute pruned-SSA, as we are only interested in live variables. The necessary liveness information is gathered in the existing liveness-pass, via fixed-point data-flow analysis, which also eliminates dead code.

3.1 Machine Instruction Constraints

The SSA construction routine receives a linked list of basic blocks containing instructions annotated with register usage and liveness information as input. The instructions are pre-colored and contain references to physical registers for special purpose registers and ABI constraints. These are not promoted to SSA variables. While this simple approach enables live range discovery, it won't be enough for other analysis and optimizations, e.g., moving liveness analysis after SSA construction. One way to remedy this, would be to promote references to physical registers to SSA variables, for which a mapping to the original register is kept. On SSA destruction, these variables are renamed to their original register name, resolving any interferences introduced by optimizations. This scheme was, however, not implemented.

Modifying instructions, e.g., single argument increment/decrement, 2-Address instructions, and conditional instructions are treated as non-kill definitions that won't introduce a new SSA name.

```
movq $1, %vI_x_0
inc  %vI_x_1
addq %vI_y_0, %vI_x_2
```

Listing 3: Wrong renaming of modified operands

Listings 3 and 4 show what would happen, if non-kill definitions were to be renamed naively. In the case of source-destination operands (modified operands), they are constraint to have the same name. If one were to treat them as normal SSA variable, this constraint

```
movq $1, %vI_x_0        # definition
addq %vI_x_0, %vI_y     # use
cmov $0, %vI_x_1        # Conditional defintion
addq %vI_x_1, %vI_z     # Error
```

Listing 4: Wrong renaming of conditional definition

would have to be modelled, e.g., by a three address instruction with explicit naming constraint. On SSA destruction, an additional copy to the newly named variable would be necessary (Listing 5).

```
movq $1, %vI_x_0
# [...]
movq %vI_x_0, %vI_x_1     # Copy to new name
addq %vI_y_0, %vI_x_1
```

Listing 5: Renaming with additional copy

Similarly, in Listing 4, the conditional move may not be executed, so using a new name afterwards would not be correct. Since there are possibly two versions of $\%vI_x$ available, something similar to a ϕ-function would be necessary. To this end, [SdF01] introduces ψ-functions to model predicated execution.

By not renaming non-kill definitions, the destruction scheme described in section 3.3 produces correct code and achieves the goal of web renaming. Any future optimization which moves code, must maintain the relative position of these not renamed instructions.

3.2 SSA construction

SSA constructions follows the algorithm by Cytron et al. [CFR+91] with added liveness checks.

A ϕ-function is needed at every control flow join point, where multiple definitions of a variable are live. Placing one for each variable in the program at each join point would obviously create huge numbers of useless ϕ-functions. To find out where to place ϕ-nodes, we need to look at the dominance property.

In a directed graph, a node n is said to dominate a node m, or $m \in Dom(n)$, if all paths from the start node to m lead through n. n is said to *strictly dominate* m if $m \in Dom(n)$ and $n \neq m$.

A variable definition in n clearly does not require a ϕ-function in m, as by the dominance property, every path to m must lead through n. Instead, ϕ-functions are required at join points just outside the region dominated by n. This *dominance frontier* of n, $DF(n)$, is defined as $m \in DF(n), \forall m, s.t.\ q \in Pred(m),\ q \in Dom(n)$ and n does not strictly dominate m.

We can then place a ϕ-function for each definition in n at the beginning of every block in $DF(n)$. Each ϕ-function is itself a definition and may induce further ϕ-functions.

The next step is *variable renaming*, in which we introduce new names for each definition. This new name consists of a *base name*, which corresponds to the original variable name and a subscripted index. Compiler generated temporary names need to be assigned a new base name as well.

For the renaming step, the *Dominator Tree* of the CFG has to be computed. The node m which *strictly dominates* and is *closest* to n, is called the *immediate dominator* of n or $IDom(n)$. The start node does not have an immediate dominator. Each node in the CFG appears in the dominator tree and an edge connects m with n if m is in $IDom(n)$.

The renaming algorithm walks the dominator tree in preorder. First, the ϕ-function definitions are renamed, then it iterates over each instruction in the block, replacing uses with the current name on the name stack and creates new names for definitions, pushing them on the name stack. Next, the ϕ-function arguments are filled in for the *successors* of the current block (in the CFG) with the current names. Then the procedure recurs into the block's children in the dominator tree. After the recursive step, the name stacks are restored to their initial state while entering this block.

3.3 SSA destruction

SSA destruction is the process of transforming from SSA form into legal machine instructions. This includes the correct elimination of ϕ-functions. Many improvements and corrections to the original destruction procedure have been proposed ([BCHS98], [BDR$^+$09]). Transforming out of SSA after optimizations have been performed requires much care, as copies have to be placed carefully for correctness. Critical edges[8] in the CFG have to be split for some of these approaches to work and algorithms have been carefully designed to avoid this costly process.

Sreedhar et al. [SJGS99] describe the difference between conventional SSA (CSSA) and transformed SSA (TSSA). Directly after SSA construction, before applying optimizations like copy propagation or code motion, the code is in CSSA. Therefore, no interferences have been introduced between SSA variables. That paper also presents an algorithm to transform TSSA back into CSSA.

Since SSA destruction is performed immediately after SSA construction in this implementation and no optimizations have been performed on the SSA form, it is still in CSSA. This allows for a straight forward and simple destruction algorithm. No copies need to be inserted and complicated control flow, such as critical edges, can be ignored.

[8]A critical node is an edge from a node with multiple successors to a node with multiple predecessors

To convert out of SSA in this simple case, vregs have to be renamed in a way, such that they form coherent webs. By performing a union-find over all ϕ-functions in the code, these webs can be built. All the sets associated with the ϕ argument names are unioned with the defined name. Purely local vregs are not included and keep their name. Each disjoint set produced this way, represents a web and all members of the set are renamed to a common name.

CHAPTER 4 Implementation

4.1 Unique Names

GHC uses so called `Uniques` as identifiers internally, which allow for easy creation of unique values by passing a unique source around. They are designed for fast comparisons and are represented by an integer. Since virtual registers are identified by Uniques, using the subscript based naming scheme common in SSA is not easily applicable. This implementation simply generates completely new Uniques, with the drawback that debugging becomes more difficult, as the connection between different Uniques is not obvious.

4.2 Code Representation

4.2.1 Liveness

Each instruction is either a machine instruction or a meta-instruction (`SPILL`, `RELOAD`). Register usage consists of a list of registers read and written per instruction. These contain both virtual and physical registers. The liveness information contains three sets of registers "born" (written to for the first time), dying because they were read for the last time and because they were written to for the last time. $Live_{In}$ sets are also provided for each block.

4.2.2 Phi-functions

For the SSA-form, LiveBasicBlocks are wrapped with SsaBasicBlocks, which also contain a list of ϕ-functions. Keeping them separate, instead of as pseudo-instructions in the instruction list, makes it easier to identify and process them. This also avoids introducing yet another intermediate representation, as adding ϕ-functions as meta-instructions to the current instruction type would mean, that they would have to be handled even after SSA destruction.

```
type LiveCmmDecl statics instr
    = GenCmmDecl
            statics
            LiveInfo
            [SCC (LiveBasicBlock instr)]

type LiveBasicBlock instr
    = GenBasicBlock (LiveInstr instr)
```

Listing 6: GHC's Live Code Representation

```
data SSABasicBlock instr
    = SSABB [PhiFun] (LiveBasicBlock instr)

data SccBits a
    = AcyclicBit a | CyclicBit Int a

type BlockLookupTable instr
    = UniqDFM BlockId (SccBits (SSABasicBlock instr))
```

Listing 7: SSA Code Representation

4.2.3 Flattening SCCs

The SSA transformation for GHC was designed to be minimally invasive and integrate into the existing backend passes. Therefore, it has to deal with the current representation for liveness annotated procedures. These don't simply contain lists of basic blocks, but instead lists of Strongly Connected Components (SCCs), where an acyclic SCC only contains one basic block and a cyclic SCC (representing a top-level loop) contains a list of basic blocks. Listing 6 shows a simplified version of the data structures, where LiveInfo contains the $Live_{In}$ sets, among other things and GenBasicBlock contains a BlockId and a list of live instructions.

The SCCs are flattened into a list and each cyclic SCC is numbered, so that consecutive `SccBits` can later be mapped to their corresponding acyclic SCC. This flattened list is then stored in a map of *basic block IDs* to `SccBits` wrapped basic blocks, to enable random access of blocks (Listing 7). The employed `UniqDFM` type is a deterministic finite map of `Uniques` to elements and guarantees maintaining insertion order for deterministic iteration, which is important for restoring the original structure. GHC's Unique Map and Set types are very efficient, as they are wrappers around *Data.IntMap* which is an implementation of big-endian patricia trees [OG98]. Most operations have a worst-case time complexity of $O(min(n, W))$, where W is the word size in bits.

4.3 Pruned SSA

Since the main focus lies on improving register allocation, it is important to remove any unproductive ϕ-function. Otherwise bogus interferences may be added to the conflict graph.

Before inserting $x_n = \phi(x_0, .., x_{n-1})$ in block b, a check is performed whether $x \in Live_{In}(b)$, i.e., this variable was live in the original program at that point. This condition is what yields *pruned-SSA* form.

An optimization was added, described in [CT11]: Each block should only be visited once per variable. To this end, a `visited` set is set to the empty set for a new variable and each block is added upon visit.

4.4 Efficient Renaming

One of the routines measured to be most resource intensive in the SSA transformation, is `rename variables`. As the `rename` routine iterates over the instructions in a block, it renames encountered uses and definitions place new names on a stack. This global name stack is needed, as the algorithm traverses the CFG depth-first, where "deeper lying" blocks may redefine a basename which is used in a direct successor of the current block.

This solution keeps one stack for each name and a list of change-sets for the current block nesting depth. As a new name for a vreg is generated, it's original name is added to the local rename set. If it was already an element of that set, the current top of the name stack is replaced, otherwise the new name pushed onto the stack. This makes sure, that the name stack size does not grow with multiple redefinitions within blocks, but is bounded by the depth of the dominator tree. After visiting all successors, the change-set for the current block contains all the redefined basenames, whose top stack elements must be removed, as the recursive function returns.

4.5 Updating Liveness Information

To avoid having to recompute liveness information after SSA destruction, it is kept up-to-date during construction and destruction.

As vregs are being renamed, so are their occurrences in the liveness information of the current instruction and the $Live_{In}$ sets. To get correct $Live_{In}$ sets, it is necessary to first process a blocks ϕ-functions, as their definitions are considered to be in $Live_{In}$. Then all names within the current block's $Live_{In}$ set are replaced with their new names from the top of the rename stacks.

During SSA destruction, $Live_{In}$ sets are updated while renaming webs. Each entry of the set is replaced with its corresponding set-name in the union-find data-structure.

CHAPTER 5 Results

The implementation was evaluated using Haskell's Nofib benchmark suite[9], which contains 154 programs organized in 7 groups. All benchmarks were compiled with the same GHC 9.1 development version containing the SSA transformation patches, activatable by a command-line switch and general optimizations (`-O2`). The hardware used is a x86_64 machine with 16GB of RAM and an AMD Ryzen 7 4700U CPU with 8 cores. Benchmarks were run five times each and average results for CPU cycles are shown.

The tables printed here focus on the *real* group, which contains larger benchmarks, aimed at being more realistic.

5.1 vreg names

The data confirms, that most programs contain disjoint live ranges with the same vreg name, as can be seen in Table 1. All but eight programs showed an increase in unique vreg after SSA destruction. The median increase for the *real* group is 4.89% and 5.44% overall, with the maximum increase being 21.04%.

[9]https://gitlab.haskell.org/ghc/nofib/ – Accessed 23.3.2021

	Before SSA	After SSA
real/anna	19409	20403
real/ben-raytrace	6431	6937
real/bspt	4019	4287
real/cacheprof	6325	6633
real/compress	798	839
real/compress2	2514	2876
real/eff/CS	87	87
real/eff/CSD	43	43
real/eff/FS	74	75
real/eff/S	10	10
real/eff/VS	93	96
real/eff/VSD	32	32
real/eff/VSM	220	220
real/fem	3375	3665
real/fluid	8740	9437
real/fulsom	4512	4812
real/gamteb	1571	1771
real/gg	3719	3858
real/grep	1387	1469
real/hidden	2441	2572
real/hpg	3297	3390
real/infer	2383	2418
real/lift	1883	1933
real/linear	3018	3172
real/maillist	187	198
real/mkhprog	564	593
real/parser	4875	5281
real/pic	1416	1515
real/prolog	1208	1243
real/reptile	3486	3631
real/rsa	157	168
real/scs	4342	4810
real/smallpt	1871	2090
real/symalg	2143	2320
real/veritas	18766	19667

Table 1: Unique vregs before SSA construction and after SSA destruction

5.2 Spill Code

GHC's graph coloring register allocator uses a very simplistic spill heuristic, namely it will always spill the longest live range. This worked unexpectedly well so far, possibly because the disjoint live ranges will be longer on average. Therefore, a modified version was tested, using the classic cost function by Chaitin, $cost(n)/deg(n)$, where $deg(n)$ is the degree of node n in the conflict graph and

$$cost(n) = \sum_{\substack{I \in Instructions \\ \text{c number of defs and uses of n in I}}} c * freq(I)$$

with $freq(I)$ being the expected execution frequency of instruction I.

Table 2 shows spill numbers for the graph coloring register allocator. *graph* is the baseline, *graph-ssa* with SSA transformation and *chaitin-ssa* with SSA transformation and the Chaitin spill heuristic. *Spills* includes `STORE` and `LOAD` instructions inserted, *reg-reg* includes all remaining register-register moves. The median reduction of spills with SSA transformation for *real* is -7.09%, of reg-reg moves -4.82%. Using the Chaitin spill heuristic, median reduction of spills is zero and of reg-reg moves -5.42%. This seems to be biased by the *real/eff* benchmarks, which didn't spill any registers to begin with. Excluding all programs in *real*, which never spilled across all test configurations, the median change of spills for *graph-ssa* becomes -21.67% and -27.50% for *chaitin-ssa*.

The median of changes in spills - excluding never spilling programs - for the whole benchmark suite is -20% for *graph-ssa* and -24.54% for *chaitin-ssa*. reg-reg moves over all programs changed by -5.4% for *graph-ssa* and -5.67% for *chaitin-ssa*.

Changes for the linear scan register allocator, which has been tuned and optimized more, are less pronounced and can be seen in Table 3. The mean change in spills is actually zero and for reg-reg moves a negligible increase of 0.34%. This does not change when excluding never spilling programs.

For the entire benchmark suite, median change in both spills and reg-reg moves is zero. When excluding never spilling programs, the median change in spills becomes -1.09%.

5.3 Runtime

To evaluate runtime performance, CPU cycles as measured by the hardware performance counters were collected. The programs in the *real/eff* subgroup were excluded, as they did not see any changes in unique names or spills. Due to their small size (e.g., real/eff/VSD has less than 30 lines) and short runtime (e.g., 12ms), any changes in runtime are most likely just noise.

Runtime performance with the graph coloring allocator shows good improvements for some benchmarks, like *real/ben-raytrace* (-8.06%), *real/hpg* (-3.35%) and *real/linear* (-3.63%). Of these programs, only *real/ben-raytrace* also exhibits a big reduction in spills, whereas *real/linear* even sees a slight *increase* in spills (but reduction in reg-reg moves of -6.62%).

Several benchmarks see increases in runtime, e.g., *real/grep* (+3.09%), *real/infer* (+6.58%) and *real/reptile* (+5.44%). A possible explanation for *real/grep* is the observed increase of 3.66% in cache misses. This benchmark even shows an improved runtime by -6.02% when using the Chaitin spill heuristic (-1.22% cache misses). *real/infer* shows performance

	graph		graph-ssa		chaitin-ssa	
	spills	reg-reg	spills	reg-reg	spills	reg-reg
real/anna	337	7077	211	6713	205	6690
real/ben-raytrace	1173	1655	767	1588	727	1589
real/bspt	73	1160	60	1130	90	1116
real/cacheprof	86	2445	72	2350	84	2343
real/compress	20	256	20	238	24	239
real/compress2	874	551	592	475	532	514
real/eff/CS	0	64	0	64	0	64
real/eff/CSD	0	22	0	22	0	22
real/eff/FS	0	28	0	26	0	26
real/eff/S	0	5	0	5	0	5
real/eff/VS	0	49	0	49	0	47
real/eff/VSD	0	10	0	10	0	10
real/eff/VSM	0	64	0	58	0	58
real/fem	253	759	187	699	174	679
real/fluid	656	1933	484	1790	402	1768
real/fulsom	0	1012	0	943	0	945
real/gamteb	349	359	236	347	225	334
real/gg	26	1430	16	1364	18	1365
real/grep	10	576	10	565	12	565
real/hidden	51	728	45	687	40	681
real/hpg	9	928	7	868	7	861
real/infer	10	772	10	748	12	748
real/lift	0	504	0	487	0	487
real/linear	191	998	197	936	46	926
real/maillist	10	94	10	87	12	87
real/mkhprog	0	241	0	229	0	225
real/parser	289	1643	143	1531	133	1526
real/pic	725	284	677	280	633	316
real/prolog	16	469	15	459	17	459
real/reptile	61	1570	50	1533	24	1532
real/rsa	10	109	10	89	12	89
real/scs	2109	1238	1780	1198	1720	1209
real/smallpt	364	546	200	566	204	562
real/symalg	37	1108	34	1057	30	1051
real/veritas	70	6500	54	6140	76	6135

Table 2: Inserted spills and remaining reg-reg moves for graph coloring register allocator

regressions for both heuristics, for *graph-ssa* possibly because of an increase in cache misses by 1.04%, whereas *chaitin-ssa* inserted two additional spills.

The average change in runtime is small. To minimize the impact of changes to very short running programs (less than a second), versus very long running programs (e.g., 58 seconds), we weight each value by the fraction of total runtime of *real* it contributes. The weighted average change in runtime for *graph-ssa* is -1.90% and -2.25% for *chaitin-ssa*.

	linear		linear-ssa	
	spills	reg-reg	spills	reg-reg
real/anna	113	9106	115	9146
real/ben-raytrace	558	2495	587	2444
real/bspt	52	1803	78	1812
real/cacheprof	72	3683	69	3697
real/compress	24	342	24	343
real/compress2	254	818	290	790
real/eff/CS	0	83	0	83
real/eff/CSD	0	30	0	30
real/eff/FS	0	33	0	33
real/eff/S	0	7	0	7
real/eff/VS	0	54	0	54
real/eff/VSD	0	14	0	14
real/eff/VSM	0	64	0	64
real/fem	92	1332	92	1329
real/fluid	353	2960	398	2939
real/fulsom	0	1245	0	1250
real/gamteb	151	583	147	585
real/gg	18	1799	18	1797
real/grep	12	715	12	718
real/hidden	40	977	40	980
real/hpg	4	1238	4	1247
real/infer	12	949	12	952
real/lift	0	629	0	634
real/linear	40	1337	27	1317
real/maillist	12	125	12	130
real/mkhprog	0	305	0	305
real/parser	72	2435	72	2449
real/pic	428	594	396	619
real/prolog	14	580	17	582
real/reptile	32	2051	36	2050
real/rsa	12	110	12	115
real/scs	1264	2261	1117	2243
real/smallpt	110	754	108	820
real/symalg	9	1498	11	1515
real/veritas	76	8490	76	8464

Table 3: Inserted spills and remaining reg-reg moves for linear scan register allocator

Chaitin's heuristic performs generally better than the simplistic live range length based one. A notable outlier is *real/reptile*. While we see a performance degradation for both heuristics, Chaitin's actually leads to a a reduction of 60% in spills. Spill placement may be at fault though, with a 3% increase in cache misses. *real/veritas*'s increased runtime can be explained by the 8.5% increase in spills.

Table 5 shows the runtime benchmarks using the linear scan allocator. There are some

	graph	std. err.	graph-ssa (rel)	std. err.	chaitin-ssa (rel)	std. err.
real/anna	4.98E9	0.30%	-0.22%	0.20%	-0.63%	0.30%
real/ben-raytrace	6.35E11	5.6e-2%	-8.06%	0.004	-7.81%	0.40%
real/bspt	3.53E11	0.10%	-0.57%	0.10%	-0.27%	0.10%
real/cacheprof	3.26E12	0.10%	-2.23%	0.30%	-4.21%	0.30%
real/compress	2.41E13	0.30%	-0.29%	0.20%	-0.17%	0.30%
real/compress2	2.84E14	0.30%	-0.50%	0.30%	-2.01%	0.20%
real/fem	3.34E15	0.20%	-0.72%	0.30%	-2.29%	0.10%
real/fluid	2.72E16	0.10%	1.12%	2.3e-2%	0.27%	0.20%
real/fulsom	1.82E17	8.4e-2%	-0.63%	0.002	-1.51%	0.60%
real/gamteb	4.83E18	0.20%	0.13%	0.20%	-1.74%	0.20%
real/gg	3.79E19	0.20%	-2.53%	0.30%	-2.89%	0.60%
real/grep	3.29E20	0.20%	3.09%	6.3e-2%	-6.02%	0.20%
real/hidden	4.43E21	0.10%	-3.11%	0.20%	1.26%	0.20%
real/hpg	3.07E22	0.50%	-3.35%	0.10%	-4.36%	5.4e-2%
real/infer	3.87E23	0.70%	6.58%	0.20%	5.91%	3.10%
real/lift	3.10E24	5.5e-2%	-0.67%	8.8e-2%	-2.09%	0.20%
real/linear	5.78E25	2.9e-2%	-3.63%	9.7e-2%	-1.72%	0.20%
real/maillist	2.86E26	0.50%	4.79%	0.80%	-0.65%	0.30%
real/mkhprog	1.66E25	1.40%	1.04%	0.90%	0.43%	0.80%
real/parser	2.76E28	0.20%	-2.44%	0.10%	1.76%	0.10%
real/pic	1.92E29	0.30%	0.49%	0.20%	-2.15%	0.60%
real/prolog	3.56E30	8.2e-2%	0.50%	0.002	-1.16%	0.10%
real/reptile	2.52E31	0.20%	5.44%	0.20%	4.19%	0.10%
real/rsa	2.76E32	5.5e-2%	0.74%	5.6e-2%	1.40%	0.10%
real/scs	2.80E33	0.20%	-0.14%	0.10%	-0.20%	0.20%
real/smallpt	1.77E36	4.6e-2%	-0.54%	6.5e-2%	-0.96%	4.3e-2%
real/symalg	3.16E35	0.20%	0.08%	2.7e-2%	0.14%	7.4e-2%
real/veritas	2.67E36	0.30%	0.53%	9.8e-2%	2.20%	0.30%
Geo. mean			-0.22%		-0.94%	
Median			-0.26%		-0.81%	
Weighted avg.			-1.90%		-2.25%	

Table 4: CPU cycles for *real* with graph coloring register allocator

improvements to *real/hidden*, *real/infer* and also to *real/hpg*, despite only small changes to number of spills. Of these, only *real/hpg* saw a drop of -4% in cache misses. The result of *real/infer* appears to be spurious though, given the high standard error for baseline runtime and the fact, that earlier runs did not show such an increase for this program.

real/cacheprof, *real/linear* and *real/mkhprog* showed significant increases in runtime. Of these, *real/cacheprof* did not see big changes in spill numbers, yet this result was reproducible. Spill-code placement may be to blame, as an increase of 1.53% in cache misses can be seen. The case is similar for *real/linear*, despite the reduction in spills. *real/mkhprog*'s result may be bogus, as this program did not spill, saw no changes to reg-reg moves, an actual *decrease* in cache misses and an elevated standard error for the *linear-ssa* measurement.

The average runtime has not changed significantly, both for *real* and the whole benchmark suite.

5.4 Compile Time

To measure the impact on compile times, the library version of Haskell's package manager *Cabal* was compiled with maximum optimizations (`-O2`) and again with the addition of SSA transformation. Containing around 100,000 lines of Haskell code, it takes sufficiently long to compile to notice any changes in compile times. The observed increase in compile times is quite considerable, at almost 30%. It's important to note though, that the implementation has not been particularly optimized.

	linear	std. err.	linear-ssa (rel)	std. err.
real/anna	5.01E9	0.20%	-0.66%	0.30%
real/ben-raytrace	5.71E11	2.0e-2%	-1.34%	1.9e-2%
real/bspt	3.52E11	0.20%	0.21%	0.20%
real/cacheprof	3.08E12	0.30%	3.85%	0.40%
real/compress	2.41E13	0.20%	0.50%	0.20%
real/compress2	2.86E14	5.5e-2%	-2.09%	0.20%
real/fem	3.29E15	7.9e-2%	-1.78%	0.40%
real/fluid	2.74E16	0.30%	-1.19%	0.10%
real/fulsom	1.77E17	6.1e-2%	2.03%	0.10%
real/gamteb	4.76E18	0.30%	1.96%	6.7e-2%
real/gg	3.38E19	0.10%	-0.03%	0.30%
real/grep	3.27E20	0.10%	1.38%	0.20%
real/hidden	4.53E21	4.1e-2%	-4.48%	0.20%
real/hpg	3.10E22	0.40%	-2.58%	6.3e-2%
real/infer	3.93E23	3.00%	-3.21%	0.20%
real/lift	3.05E24	8.8e-2%	1.44%	0.10%
real/linear	5.50E25	5.5e-2%	4.10%	0.10%
real/maillist	2.82E26	0.50%	1.11%	0.40%
real/mkhprog	1.64E25	0.70%	3.29%	1.30%
real/parser	2.76E28	0.20%	0.60%	5.8e-2%
real/pic	1.94E29	0.50%	-1.09%	0.40%
real/prolog	3.56E30	0.20%	-2.23%	0.20%
real/reptile	2.59E31	0.30%	0.12%	0.10%
real/rsa	2.77E32	7.7e-2%	0.62%	6.4e-2%
real/scs	2.74E33	0.10%	-0.34%	0.40%
real/smallpt	1.81E36	0.10%	-0.57%	0.20%
real/symalg	3.16E35	7.0e-2%	0.08%	1.7e-2%
real/veritas	2.69E36	0.10%	1.97%	0.30%
Geo. Mean			0.04%	
Median			0.10%	
Weighted avg.			-0.56%	

Table 5: CPU cycles for *real* with linear scan register allocator

CHAPTER 6

Conclusions and Future Work

Static Single Assignment form intermediate representations are a fundamental technique in modern compiler construction, providing a sparse representation of data-flow information and enabling many optimizations in an efficient manner. Even at a late state in the compilation pipeline, when dealing with native code, SSA can be very beneficial.

Adding SSA transformation to GHC's native code generator effectively performs renumbering of disjoint live ranges, facilitating register allocation and leading to reduced spill code insertion. This is especially pronounced for the graph coloring register allocator. Furthermore it enables future optimizations, such as sparse conditional constant propagation or live range splitting in the register allocator.

However, more work needs to be done to increase efficiency of the implementation. Using the same base algorithm by Cytron et al., semi-pruned SSA form could be computed more cheaply, followed by liveness analysis on SSA form. The approach by Braun et al. [BBH$^+$13] also promises to be more efficient, not requiring prior liveness analysis or computation of dominance frontiers. Another advantage is the fact, that it produces pruned SSA directly.

To enable further optimizations, it may also become necessary to model machine instruction constraints.

Acronyms

ABI	Application Binary Interface. 9
CFG	Control-flow graph. 2, 6, 11, 14
GHC	Glasgow Haskell Compiler. vii, 1, 2, 12–14, 16, 23
IR	intermediate representation. 2
ISA	Instruction Set Architecture. 6
NCG	Native Code Generator backend. vii, 2, 4
SCC	Strongly Connected Component. 13, 14
SCCs	Strongly Connected Components. 13, 14
SSA	Static Single Assignment. vii, 4–15, 17, 21, 23
vreg	virtual register. 3, 4, 12, 14, 15

Bibliography

[BBH^{+}13] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. Simple and efficient construction of static single assignment form. In Ranjit Jhala and Koen De Bosschere, editors, *Compiler Construction*, pages 102–122, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[BBN$^+$18] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL):5:1–5:29, 2018.

[BCHS98] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exp.*, 28(8):859–881, 1998.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[BDR$^+$09] Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoît Dupont de Dinechin, and Christophe Guillon. Revisiting Out-of-SSA translation for correctness, code quality and efficiency. In *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009*, pages 114–125. IEEE Computer Society, 2009.

[Bri14] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, 2014.

[CAC$^+$81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Comput. Lang.*, 6(1):47–57, 1981.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In David S. Wise, editor, *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, pages 55–66. ACM Press, 1991.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, 1990.

[CT11] Keith D. Cooper and Linda Torczon. *Engineering a Compiler (Second Edition).* Morgan Kaufmann, 2011.

[dD14] Benoît Dupont de Dinechin. Using the SSA-Form in a code generator. In Albert Cohen, editor, *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and*

Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings, volume 8409 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2014.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[Hec77] Matthew S Hecht. *Flow analysis of computer programs.* Elsevier Science Inc., 1977.

[HGG06] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In Alan Mycroft and Andreas Zeller, editors, *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings*, volume 3923 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2006.

[HHJW07] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In Barbara G. Ryder and Brent Hailpern, editors, *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, pages 1–55. ACM, 2007.

[Jon92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-Machine. *J. Funct. Program.*, 2(2):127–202, 1992.

[JRR99] Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. C–: A portable assembly language that supports garbage collection. In Gopalan Nadathur, editor, *Principles and Practice of Declarative Programming, International Conference PPDP'99, Paris, France, September 29 - October 1, 1999, Proceedings*, volume 1702 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 1999.

[JW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 71–84. ACM Press, 1993.

[LG99] Allen Leung and Lal George. Static single assignment form for machine code. In Barbara G. Ryder and Benjamin G. Zorn, editors, *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*, pages 204–214. ACM, 1999.

[OG98] Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *In Workshop on ML*, pages 77–86, 1998.

[PM04] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004.

[SCJD07] Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In François Pottier and George C. Necula, editors, *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, pages 53–66. ACM, 2007.

[SdF01] Artour Stoutchinin and François de Ferrière. Efficient static single assignment form for predication. In Yale N. Patt, Josh Fisher, Paolo Faraboschi, and Kevin Skadron, editors, *Proceedings of the 34th Annual International Symposium on Microarchitecture, Austin, Texas, USA, December 1-5, 2001*, pages 172–181. ACM/IEEE Computer Society, 2001.

[Sin18] Jeremy Singer. Introduction. In *Single Static Assignment Book*, chapter 1. 2018. INRIA.

[SJGS99] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings*, volume 1694 of *Lecture Notes in Computer Science*, pages 194–210. Springer, 1999.

[TC10] David A. Terei and Manuel M. T. Chakravarty. An LLVM backend for GHC. In Jeremy Gibbons, editor, *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, pages 109–120. ACM, 2010.

[WF10] Christian Wimmer and Michael Franz. Linear scan register allocation on SSA form. In Andreas Moshovos, J. Gregory Steffan, Kim M. Hazelwood, and David R. Kaeli, editors, *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*, pages 170–179. ACM, 2010.