Language Support for Programming High-Performance Code

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
an der Fakultät für Mathematik und Informatik der
Universität des Saarlandes

eingereicht von

Roland Leißa

Saarbrücken, 2017

UNIVERSITÄT
DES
SAARLANDES

Day of Colloquium	2018-02-22
Dean of the Faculty	Univ.-Prof. Dr. Frank-Olaf Schreyer
Chair of the Committee	Prof. Dr. Gert Smolka
First Reviewer	Prof. Dr. Sebastian Hack
Second Reviewer	Prof. Dr. Philipp Slusallek
Third Reviewer	Prof. Dr. Christophe Dubach
Academic Assistant	Dr. Richard Membarth

Abstract

Nowadays, the computing landscape is becoming increasingly heterogeneous and this trend is currently showing no signs of turning around. In particular, hardware becomes more and more specialized and exhibits different forms of parallelism. For performance-critical codes it is indispensable to address hardware-specific peculiarities. Because of the halting problem, however, it is unrealistic to assume that a program implemented in a general-purpose programming language can be fully automatically compiled to such specialized hardware while still delivering peak performance.

One form of parallelism is *single instruction, multiple data (SIMD)*. Part I of this thesis presents *Sierra*: an extension for C++ that facilitates portable and effective SIMD programming.

Part II discusses *AnyDSL*. This framework allows to embed a so-called *domain-specific language (DSL)* into a *host language*. On the one hand, a DSL offers the application developer a convenient interface; on the other hand, a DSL can perform domain-specific optimizations and effectively map DSL constructs to various architectures. In order to implement a DSL, one usually has to write or modify a compiler. With AnyDSL though, the DSL constructs are directly implemented in the host language while a *partial evaluator* removes any abstractions that are required in the implementation of the DSL.

Zusammenfassung

Die Rechnerlandschaft wird heutzutage immer heterogener und derzeit ist keine Trendwende in Sicht. Insbesondere wird die Hardware immer spezialisierter und weist verschiedene Formen der Parallelität auf. Für performante Programme ist es unabdingbar, hardwarespezifische Eigenheiten zu adressieren. Wegen des Halteproblems ist es allerdings unrealistisch anzunehmen, dass ein Programm, das in einer universell einsetzbaren Programmiersprache implementiert ist, vollautomatisch auf solche spezialisierte Hardware übersetzt werden kann und dabei noch Spitzenleistung erzielt.

Eine Form der Parallelität ist „single instruction, multiple data (SIMD)“. Teil I dieser Arbeit stellt *Sierra* vor: eine Erweiterung für C++, die portable und effektive SIMD-Programmierung unterstützt.

Teil II behandelt *AnyDSL*. Dieses Rahmenwerk ermöglicht es, eine sogenannte *domänenspezifische Sprache (DSL)* in eine *Gastsprache* einzubetten. Auf der einen Seite bietet eine DSL dem Anwendungsentwickler eine komfortable Schnittstelle; auf der anderen Seiten kann eine DSL domänenspezifische Optimierungen durchführen und DSL-Konstrukte effektiv auf verschiedene Architekturen abbilden. Um eine DSL zu implementieren, muss man gewöhnlich einen Compiler schreiben oder modifizieren. In AnyDSL werden die DSL-Konstrukte jedoch direkt in der Gastsprache implementiert und ein *partieller Auswerter* entfernt jegliche Abstraktionen, die in der Implementierung der DSL benötigt werden.

Preface

Since the beginnings of integrated circuits, the transistor density has approximately doubled every two years—as Moore's famous law has correctly predicted so far [Moo00]. Over decades, this development had gone hand in hand with an exponential increase in processor speed. In the middle of the 2000s, this process stagnated because it was no longer possible to dissipate the heat of the processor. This means that advances in the microprocessors' manufacturing process do not lead to an acceleration of *existing* software anymore as processors hardly become faster.

However, it is generally assumed that Moore's law will continue to be valid for some time [Kum15]: the additional transistors are used for special processors that are particularly energy-efficient for certain computing models. For example, GPUs are tailor-made for data-parallel problems; Xeon Phis have many simple cores with very wide SIMD units. But using this heterogeneity effectively burdens the programmer with major problems: A high-performance implementation of an algorithm for one architecture can differ significantly from that for another architecture. Even on today's systems one can rarely achieve peak performance with a "textbook implementation" of an algorithm. The properties of the hardware (memory hierarchy, memory layout, SIMD instructions, etc.) are too performance-critical to be disregarded. However, automatically performing many of the necessary code transformations exceeds the ability of today's compilers.

A promising way out of this problem are *DSLs*. DSLs offer *language constructs* which reflect the essential abstractions of a domain-specific algorithm. A compiler that has been especially built for a DSL and *understands* its abstractions can then generate high-performance code. On the downside, the DSL developer needs to design a new language and craft a new compiler for each domain. This is neither practical nor is it possible to combine DSLs in this way.

Part I of this thesis presents *Sierra—a SIMD extension for C++*. Sierra is a domain-specific extension, which gives the programmer fine-grained control over *where* and *how* his program is vectorized. This vectorized program is suitable for efficient execution on the SIMD unit.

Part II discusses *AnyDSL*—a general approach to DSLs: Instead of writing or extending a compiler, a DSL is *embedded* into a *host language*. This technique makes it possible to *seamlessly* add and combine DSLs while inheriting the host language's syntax and type system. The DSL designer implements the domain-specific extensions directly in the host language while a *partial evaluator* instantiates any abstractions of this implementation. Furthermore, the DSL designer can map code to different hardware accelerators (SIMD units, multi-core processors, GPUs) by passing the code in question to built-in higher-order functions. This allows the designer to implement a DSL *as a library* while supporting *heterogeneous* hardware without having to write her own compiler.

Contents

List of Figures

List of Listings

Part I.

Sierra
A SIMD Extension for C++

We should forget about small efficiencies, say about 97% of the time [...]. Yet we should not pass up our opportunities in that critical 3%.

Donald Knuth,
Structured Programming With Go To Statements

1 Introduction

SIMD instructions [Fly72] allow to simultaneously process multiple data in one operation (see right-hand side of Figure 1.1). SIMD hardware usually provides a special SIMD register file along with appropriate instructions to operate on these registers. We call the number of elements, which fit into a SIMD register, the *vector length*. Data-parallel workloads can be sped up by a factor of that vector length. Usually, data structures—and hence the core algorithms working on them—must be adapted in order to exploit SIMD effectively [e.g. Wal+01; ZR02]. For this reason, it is unlikely that automatic vectorization techniques can catch up with languages that directly support data parallelism.

For example, the following `Vec3` type is used in many graphics applications:

```
struct Vec3 { float x, y, z; };
```

It is inefficient for SIMD hardware to fetch several logically consecutive `x`-elements in parallel when using a traditional array of structures (AoS) (see Figure 1.2a) because the `x`-elements are physically scattered in memory. When using the structure of array (SoA) layout (see Figure 1.2b) we should exploit efficient vector loads instead. However, when looping over an SoA that needs to process `x`-, `y`-, and `z`-elements, the loop must maintain three pointers: one for each element. Each iteration increments these pointers by the underlying vector length. Yet another alternative is to inflate the original `Vec3` type by the desired vector length and group this new type in an array. This layout is called *hybrid SoA* [Int16, §3.6.6] or array of structures of arrays (AoSoA) (see Figure 1.2c).[1] Now, each group of `x`-, `y`-, and `z`-elements lies at a constant offset within one inflated `Vec3` instance. Hence, said loop must only maintain one pointer. Moreover, this layout

[1] Confusingly, this layout is often just called SoA in literature, too.

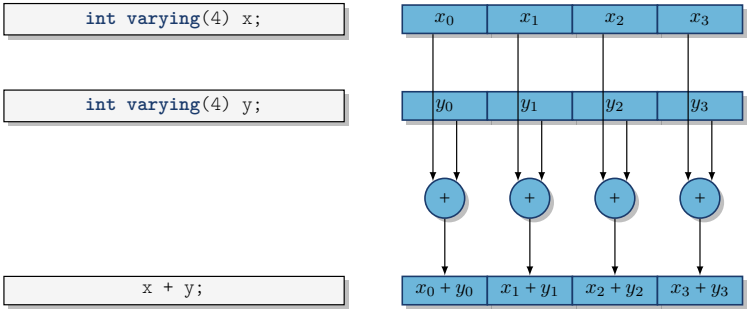


Figure 1.1.: Varying types and operations in Sierra

provides better data locality because all data needed within one iteration lies in one chunk of memory.

After the data is properly laid out, the programmer has to write the actual code. This is arguably the biggest challenge in SIMD programming. Often, operations should only be performed on *part* of a SIMD vector. Thus, operations must be *masked*. Manually writing this masking code is extremely cumbersome and error-prone as it enforces a very low-level programming style (see Section 1.1.2). Furthermore, how should the programmer design his program so that it scales to different vector lengths? After all, different instruction set architectures (ISAs) support different vector lengths.

1.1. Sierra

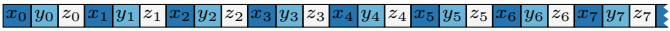
Sierra is a proposed extension for C++. This extension gives the programmer tools to vectorize both data and code in a semi-automatic way.

1.1.1. Vectorizing Data

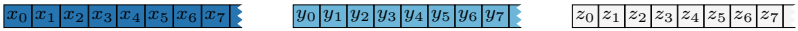
Sierra enriches C++ with the `varying` type constructor. It can be used to vectorize data types.[2] Standard operators are overloaded to work on vectors:

```
int varying(4) a = {0, 1, 2, 3};
int varying(4) b = {2, 4, 8, 10};
int varying(4) c = a + b; // 2, 5, 10, 13
```

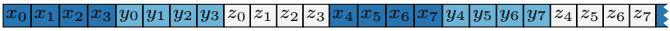
[2]The name `varying` is inspired by RenderMan (see Section 2.2.5).



(a) Array of Structures (AoS)



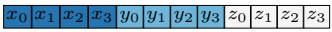
(b) Structure of Arrays (SoA)



(c) Array of Structure of Arrays (AoSoA)

Figure 1.2.: Different array layouts with vector length 4 for a `struct` with three members: `x`, `y` and `z`.

Moreover, the programmer can use this keyword to recursively inflate derived types:

> **Example 1.1** (Vectorized `Vec3`)
> The type `Vec3` `varying`(4) has the following layout:
>
> 
>

These types can be used as building blocks to create more advanced data structures like the aforementioned SoA or AoSoA layout. Additionally, the `varying` type constructor can be tightly integrated with C++ templates. This allows a C++ programmer to create sophisticated, generic, SIMD-friendly data structures.

1.1.2. Vectorizing Code

Sierra's type system does not only allow scalars as a controlling condition of control-flow constructs but also vectors. Sierra statically marks all statements that are control-dependent on a vectorial condition to be executed in *SIMD mode* as opposed to the usual *scalar mode*. The execution model of SIMD mode maps each element in a SIMD vector—also called *SIMD lane*—to one thread. This gives the programmer the illusion that different SIMD lanes are executed on different control-flow paths.

Example 1.2 (Vectorized Control-Flow)
As the condition `v < 3` in Listing 1.1 is of type `bool varying(4)`, Sierra enters *SIMD mode of length 4*. Since the condition only holds for the first and fourth SIMD lane, `v += 2` is only performed for the first and fourth element of `v`. We say the first and the fourth *lanes* are *active*. The other two lanes are *inactive*. Contrarily, `v -= 3` is only applied to the second and third element of `v`.

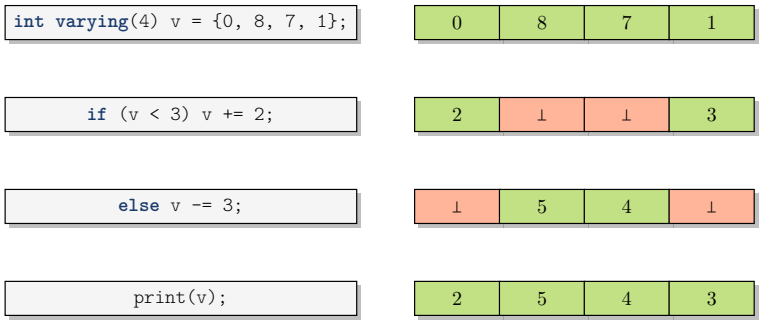
The implementation must take care that all SIMD lanes get their opportunity to be executed in the code blocks in which they are active. Conversely, all inactive lanes must not produce side-effects. To do this, the compiler must linearize the control flow and emulate it by inserting bit operations and masking instructions (see Listing 1.2).

Doing this by hand is a complicated matter. The programmer must use so-called *compiler intrinsics*: compiler-known functions that directly map to a machine instruction. These intrinsics severely degrade the readability of the code because control statements are replaced by clumsy mask-handling code as Listing 1.2 demonstrates. The burden of this conversion increases even more drastically when complex control flow like nested loops and unstructured control flow must be emulated. Furthermore, intrinsics directly expose a concrete ISA to the programmer. Accordingly, intrinsics code is not only inherently unportable between different processor architectures but also within different revisions of the ISA *within* the same processor family.

1.1.3. What Sierra is and is not

Sierra is *not* a proposal for the C++ standard.[3] Due to the complexity of C++, a proposal would be hundreds of pages long because the `varying` type constructor and SIMD mode virtually interact with all aspects of the C++ language. The ISO C++ committee would, in any case, likely reject this proposal because of its length. Instead, Sierra should be viewed as an extension similar to OpenMP [Ope13], Cilk[4], OpenACC [Ope15] or SYCL [Khr15]. This allows Sierra to cherry-pick a well-understood subset of

[3] see `https://isocpp.org/std/submit-a-proposal`
[4] see `https://www.cilkplus.org/cilk-history`



Listing 1.1.: Automatic masking in vectorized control flow

```
int4 v = {0, 8, 7, 1};
bool4 mask = lt4(v, broadcast4(3));
int4 v_t = add4(v, broadcast4(2));
int4 v_f = sub4(v, broadcast4(3));
int4 v_b = blend4(mask, v_t, v_f);
print(v_b);
```

Listing 1.2.: Using (pseudo)-intrinsics to implement Listing 1.1

C++ to define its semantics on. Complex features like exception handling or multiple inheritance are simply declared incompatible with Sierra. This does not mean that these features are completely denied to programmers. Only Sierra-specific features cannot be mixed with these incompatible features. Moreover, this thesis is not a technical manual for Sierra.

1.2. Case Study: A Volume Ray Caster in Sierra

In order to better understand the core concepts behind Sierra, we demonstrate how to implement a volume ray caster [AH02]. Volume ray casting is a technique to visualize a 3D volumetric data set by shooting rays from the camera through each pixel of the image to be rendered. The renderer marches along each ray that hits the volume and accumulates encountered

```
void render(float volume[], float image[], /*...*/) {
  for (int y = 0; y < image_height; ++y) {
    for (int x = 0; x < image_width; ++x) {

      auto ray = generate_ray(x, y, /*...*/);
      auto result = raymarch(volume, ray, /*...*/);
      image[y * image_width + x] = result;
    }
  }
}

float
raymarch(float volume[], Ray& ray, /*...*/) {
  float rayT0, rayT1;
  if (!intersect(ray, bounding_box, rayT0, rayT1))
    return 0.f;
  // intersect initializes rayT0, rayT1

  // radiance along the ray
  float result = 0.f;

  // induction variables
  auto pos = ray.dir*rayT0 + ray.origin;
  auto t = rayT0;

  while (t < rayT1) {
    auto d = density(pos, volume, /*...*/);

    // terminate on high attenuation
    auto atten = /*...*/;
    if (atten > THRESHOLD)
      break;

    auto light = compute_lighting(/*...*/);
    result += light * /*...*/;
    pos += /*...*/;
    t += /*...*/;
  }

  return gamma_correction(result);
}
```

Listing 1.3.: Volume renderer in C++

```
void render(float volume[], float varying(L) image[], /*...*/) {
  for (int y = 0; y < image_height; ++y) {
    for (int xx = 0; xx < image_width/L; ++xx) {
      auto x = xx*L + seq<L>();
      auto ray = generate_ray(x, y, /*...*/);
      auto result = raymarch(volume, ray, /*...*/);
      image[y * image_width/L + xx] = result;
    }
  }
}

float varying(L)
raymarch(float volume[], Ray varying(L)& ray, /*...*/) {
  float varying(L) rayT0, rayT1;
  if (!intersect(ray, bounding_box, rayT0, rayT1))
      return 0.f;
  // intersect initializes rayT0, rayT1

  // radiance along the ray
  float varying(L) result = 0.f;

  // induction variables
  auto pos = ray.dir*rayT0 + ray.origin;
  auto t = rayT0;

  while (t < rayT1) {
    auto d = density(pos, volume, /*...*/);

    // terminate on high attenuation
    auto atten = /*...*/;
    if (atten > THRESHOLD)
      break;

    auto light = compute_lighting(/*...*/);
    result += light * /*...*/;
    pos += /*...*/;
    t += /*...*/;
  }

  return gamma_correction(result);
}
```

Listing 1.4.: Volume renderer in Sierra. Differences are highlighted.

voxel data. We need—in addition to the Vec3 data type—a Ray type:

```
struct Ray {
    Vec3 origin;
    Vec3 dir;
};
```

First, we sketch the ordinary scalar version (see Listing 1.3). Then, we demonstrate how to use Sierra for vectorization (see Listing 1.4).

1.2.1. A Scalar Volume Ray Caster

Render

The render function sets up a loop nest, which iterates over all pixels of the image buffer. Each iteration generates an appropriate ray for the current pixel and invokes raymarch. This function returns the current pixel's brightness, which is assigned to the current pixel.

Ray March

The subroutine raymarch calls intersect in order to determine whether ray hits the bounding box of the volume at all. If this is the case, intersect initializes the start and end parameters rayT0 and rayT1 for ray. A loop traverses ray via its parameter t. Each iteration fetches a density for the current position pos in the volume. As an optimization, we introduce an early termination condition: If the current radiance result is greater than a certain THRESHOLD, the loop will be aborted because any further accumulation will not significantly contribute to the final brightness. Next, we compute lighting for the current position and accumulate that value in the radiance result. Finally, we return the computed radiance after applying a gamma_correction.

1.2.2. A Vectorized Volume Ray Caster

In order to exploit SIMD hardware, we simultaneously shoot L rays through the volume. We use Sierra's type constructor **varying**(L) to create SIMD-friendly variants of the original data types Vec3 and Ray. The type Ray **varying**(L) consists of two Vec3 **varying**(L) data types. Further-

more, the `image` buffer is composed of an array of **`float varying`**`(L)`.[5] Note that the program is parametric in the vector length `L`.

Render

The `render` function vectorizes along `L` consecutive pixels in `x`-direction. Each iteration creates a vector `x = {xx*L+0, xx*L+1, ..., xx*L+L-1}`. Thus, in the case of vector length 4, `x` is `{0, 1, 2, 3}` in the first iteration, `{4, 5, 6, 7}` in the second, `{8, 9, 10, 11}` in the third and so on.

The loop body invokes `generate_ray`. In contrast to the scalar version, the function expects an **`int varying`**`(L)` as `x` value and returns a `Ray` **`varying`**`(L)` as result. We feed this vectorial ray to the vectorial version of the `raymarch` function, which in turn produces a **`float varying`**`(L)`. This is stored via an efficient vector store in the `image` buffer.

Ray March

The function `raymarch` works on `L` rays simultaneously and returns `L` results. The function `intersect` now expects a `Ray` **`varying`**`(L)` and returns a **`bool varying`**`(L)`. As the condition in the *if-statement* is vectorial, Sierra only continues executing SIMD lanes that actually hit the volume. Similarly, the condition of the *while-statement* is vectorial. The loop runs until all lanes become inactive. Some lanes may terminate earlier than others and consequently become inactive because of the *break-statement*.

For this reason it is a good idea to increase the likelihood that a vector of rays shares the same control-flow. We can achieve this by vectorizing in small tiles of the target image rather than along the x-direction. For example, in the case of vector length 4, we could vectorize for each 2×2 pixel block.

1.2.3. Comparison of both Versions

The scalar and the vectorial versions are syntactically very similar. For the most part, they just differ in typing. The use of **`auto`** even hides many of these differences. Merely the initial loop, which sets up the vectorization,

[5] For the sake of simplicity, we assume that `image_width` is a multiple of `L`. If we also want to support other image widths, we either need to process the border area in a scalar fashion or mask it properly with an appropriate vectorial *if-statement*.

has to be worked on more carefully by the programmer. Thus, it is not much effort to port a scalar program to a vectorial one.

In particular, instantiating the vectorial program with `L = 1` generates the scalar version of the program: The conditions in the *if-* and *while-statements* become scalar again, which in turn triggers usual scalar semantics of C++. Chapter 6 investigates the performance of several instantiations.

1.3. Contributions

After the discussion of related work in Chapter 2, this part of the thesis makes the following contributions:

- Chapter 3 presents Sierra in more detail and discusses the issues of integrating Sierra with the rest of C++.
- Chapter 4 introduces Imp: a small imperative language. We present Imp's semantics and prove its type system sound. Then, we show how to extend Imp with SIMD types and SIMD mode—calling the new language VecImp. We prove VecImp's type system sound, too. Finally, we sketch a language PolyVecImp where `varying` annotations are inferred in a semi-automatic way.
- Chapter 5 discusses how to implement the execution model of SIMD mode. We describe how to generate code from the abstract syntax tree (AST) of a VecImp program.
- Our experiments in Chapter 6 demonstrate that our prototype implementation is able to speed up applications by 2x to 5x on SSE and by 2.5x to 7x on AVX compared to their scalar counterparts.

1.4. Publications

The work in this part is based upon the following publications:

- Roland Leißa, Sebastian Hack, and Ingo Wald. "Extending a C-like language for portable SIMD programming". In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA,*

February 25-29, 2012. 2012, pp. 65–74. DOI: 10.1145/2145816.2145825.

- Roland Leißa, Immanuel Haffner, and Sebastian Hack. "Sierra: a SIMD extension for C++". In: *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing, WPMVP 2014, Orlando, Florida, USA, February 16, 2014*. 2014, pp. 17–24. DOI: 10.1145/2568058.2568062.
- Immanuel Haffner. "Sierra: A SIMD Extension for C++". Advisors: Sebastian Hack and Roland Leißa. Bachelor's thesis. Saarland University, Apr. 1, 2015

Some paragraphs appear verbatim in this thesis. This applies in particular to Chapters 2 and 6.

Nenn' es dann wie du willst,
Nenn's Glück! Herz! Liebe! Gott!
Ich habe keinen Namen
Dafür! Gefühl ist alles;
Name ist Schall und Rauch,
Umnebelnd Himmelsgluth.

Johann Wolfgang von Goethe, Faust I

2 Related Work

Vectorizing compilers have a long history. Traditional approaches try to vectorize loops or straight-line code. As these approaches to this day have difficulties in utilizing the full potential of SIMD hardware, language designers have tried to support SIMD directly in the programming language. This is also Sierra's approach. Finally, whole-function vectorization (WFV) instantiates a whole function several times such that each instance can be executed in a different SIMD lane. In the following, each section deals with one of these approaches.

2.1. Automatic Vectorization Techniques

2.1.1. Loop Vectorization

Allen et al. [All+83; AK87] present a source-to-source compiler that tries to translate FORTRAN-77-style loop nests to FORTRAN-8x-style array statements.[1] Starting from the innermost loop, their loop vectorization instruments a loop dependence analysis in order to investigate which loops are suitable for vectorization at all. Then, the loop body's control flow is converted to data flow via *if-conversion*. This means that conditional branches are eliminated by guarding statements with an `if`. The `if`'s guard boolean condition now represents the former control flow.

An alternative is *outer loop vectorization* using a so-called *unroll-and-jam* technique [Ngo95; AK01; NZ08]: A chosen outer loop is unrolled several times while the resulting loop bodies are re-fused ("jammed"). There is a good chance that the instructions stemming from the same instruction in the original version can be grouped into SIMD instructions.

[1] FORTRAN 8x was a draft of the evolving FORTRAN 90 standard. See Section 2.2.2 for a discussion of array programming.

Other work on loop vectorization also considers data alignment, reductions [NRZ06] and interleaved data accesses [NH06]. Furthermore, the polyhedral model [Fea91; Fea92a; Fea92b]—a powerful mathematical loop analysis framework using Presburger sets—has also been instrumented for loop vectorization [Tri+09; Nuz+11]

The volume renderer from Section 1.2 *could* be automatically vectorized by vectorizing the inner loop in `render`. Admittedly, an interprocedural analysis would have to find that each iteration is independent of the others. With the presence of a complex call graph, aliasing pointers and incoherent memory access patterns (gathers—see Section 3.1.2), this is unrealistic because static analyses are undecidable due to the halting problem.

2.1.2. Superword Level Parallelism

Vectorization on straight-line code exploits superword-level parallelism (SLP): The compiler tries to merge several scalar operations into a vector operation. This can be done on a per-basic-block level [LA00] or in the presence of control flow [SHC05]. SLP algorithms will usually give up if the exact number of needed instructions cannot be fed into the SIMD lanes. *Padded SLP* tries to overcome this limitation by injecting redundant instructions [PMJ15].

The work in this field is orthogonal to the work in this part of the thesis. Vectorizing straight-line code can achieve a nice performance bonus for scalar code. On the other hand, current ISAs often take a while to "warm up" the SIMD unit—an effect that we also observed in our experiments (see Chapter 6). *Throttled SLP* uses a cost model in order to estimate whether SLP vectorization is actually worthwhile at all [PJ15].

2.2. Support in Programming Languages

2.2.1. Short Vectors

Many C/C++ compilers provide short vector data types—similar to Sierra's `varying` types. The compiler can easily map operations on these types to the hardware. Furthermore, such compilers provide ISA-specific *intrinsics* (see Section 1.1.2). Some libraries like Boost.SIMD [Est+14] or the *Generic SIMD Library* [Wan+14] wrap these functionalities in portable libraries for

easier use. Nonetheless, the programmer still needs to manually convert control flow to data flow and perform masking by himself.

2.2.2. Array Programming

Some languages like APL [FI73], Vector Pascal [Coc02], MatLab[2], Octave[3], or FORTRAN 90, and language extensions like Intel® Array Building Blocks (ArBB) [New+11][4] (formerly known as Ct [Ghu+07]) allow operations not only on scalars but also on arrays. This is similar to the way Sierra overloads standard operators to work on `varying` types. But, in contrast to Sierra, where the programmer works with short vectors of known length, the programmer typically operates with large arrays of unknown length. The compiler automatically generates an appropriately vectorized loop.

Example 2.1
Suppose `a` and `b` are arrays of size `N` with element type `int` and `c` is a scalar of type `int`:

```
a = b + c
```

It is straightforward to generate the following vectorized loop using vector length `L`:

```
for i = 0 to N step L
    a[i..L-1] = b[i..L-1] + [c, ..., c]
next
```

While this programming works well in the domain of vector and matrix computations, other programming patterns are not easily mappable to this paradigm. For example, the volume renderer from Section 1.2 does not use such patterns at all. In addition, vectorization only works for arithmetic types and not, for instance, for an array of `Rays`. Finally, each array statement conceptually introduces its own loop. In order to minimize the overhead caused by these loops, it is a good idea to fuse them. Fusing

[2] see `https://www.mathworks.com/help/matlab/matlab_prog/vectorization.html`
[3] see `https://www.gnu.org/software/octave/doc/interpreter/Basic-Vectorization.html`
[4] ArBB actually is a deeply embedded DSL using C++ as host language (see Section 10.2.3).

these loops, however, is not always possible if they are interrupted by other statements that produce side-effects.

2.2.3. Cilk Plus

Aside from providing facilities for multithreaded parallel computing, Intel® Cilk™ Plus [Int13] supports SIMD in two ways.

First, a loop may be annotated by `#pragma simd`. This allows the compiler to vectorize the loop even if the compiler cannot guarantee to preserve the semantics of the original scalar program. However, the compiler will not reorganize the program's data structures as Sierra's `varying` type constructor does. Instead, data may be reordered on the fly. Furthermore, calls to functions compiled in other translation units cannot be vectorized. Usually, the compiler will try to inline functions into the loop's body so that vectorization does not have to be performed in an interprocedural manner.

Second, Cilk Plus provides special constructs to deal with arrays in a convenient way much like array programming discussed earlier. Cilk Plus can partially mimic a Sierra type `T varying(L)`. As long as `T` is an arithmetic type, the Cilk Plus type `T[L]` behaves similar. However, Cilk Plus does not support automatic masking. This makes short vectors in Cilk Plus less useful. On the other hand, Sierra can mimic Cilk Plus's long vectors. One way would be template specializations for `std::valarray<int>`, `std::valarray<float>`, and so forth, which internally work on `int varying(L)*` or `float varying(L)*`, respectively.

2.2.4. OpenMP

OpenMP 4.0 [Ope13] also introduces an annotation to mark loops as vectorizable. Additionally, functions can be declared with `#pragma declare simd`. This allows OpenMP to call a vectorized version of a function from within a vectorized loop. A number of clauses control the behavior of the vectorized version. The clause `simdlen(L)` corresponds to setting all parameters as `varying(L)` types in Sierra. Additionally, parameters can be declared as `uniform` like in Sierra. The clause `inbranch` is similar to `simd(L)`. In Sierra however, the programmer gets more fine-grained control over the vectorization lengths, as Sierra allows mixing of vector lengths to a certain degree. The OpenMP specification is intentionally unclear about how exactly types are transformed. Since OpenMP 4.0 has been released after

Intel® single program, multiple data (SPMD) compiler (ISPC) [PM12] and our initial work on VecImp [LHW12], OpenMP 4.0 was likely influenced by these works.

2.2.5. Data-Parallel Languages

Data-parallel languages originate from shading languages—RenderMan [HL90] being one of the first. RenderMan also pioneered the concept of `uniform` and `varying` variables. Modern shading languages like Cg [Mar+03], CGiS [FLS04], GLSL [Khr13], or HLSL [SW05], and also general-purpose data-parallel languages like CUDA [NVI17], OpenCL [Khr12], IVL (see Section 6.1), or ISPC [PM12] still follow the same programming model: The programmer basically writes a scalar program. The compiler instantiates the program n times to run it simultaneously on n computing resources. When those computing resources are only SIMD processors, all simultaneously running processes run in lockstep. Program instances may run asynchronously when the compiler also leverages multithreading. Therefore, the programmer has to use barrier synchronization in order to communicate across program instances [Shi+08].

Sierra only deals with SIMD and therefore does not need any synchronization mechanisms. Supporting asynchronous threads is orthogonal to Sierra's approach. For example, a Sierra programmer could use POSIX threads or OpenMP's threading facilities to also parallelize her program.

Sierra borrows the idea to overload control-flow constructs for vectors from ISPC [PM12] and IVL/VecImp [LHW12]. IVL/VecImp and ISPC were developed simultaneously and influenced each other's design. New to Sierra's approach is that the program starts off in scalar mode. The programmer explicitly triggers vectorization by using vector types. This allows a Sierra programmer to mix various vector lengths to a certain extent. In contrast to Sierra, an ISPC or IVL programmer has to agree on a global vectorization length per translation unit. Therefore, Sierra does not need a special kernel language, which then gets plugged into the host language. We believe that this is a major obstacle in practice to adopt languages like OpenCL, CUDA, ISPC or IVL.

2.2.6. Domain-Specific Languages

A DSL is a language that is exactly tailored for a particular programming domain. Actually, array programming (see Section 2.2.2) or Sierra's SIMD extension can be viewed as domain-specific programming. As we have discussed in Section 2.1.1, the difficult part in automatic vectorization is to actually *prove* that several consecutive iterations are independent of each other. But if the compiler simply *knows* from the problem domain that this is the case, code can be vectorized with tools like WFV (see Section 2.3). For example, HIPA^{cc}—a DSL for image processing—uses a vectorization technique similar to WFV [Rei+17]. However, HIPA^{cc}'s vectorization is simplified because HIPA^{cc} leverages domain knowledge. Diderot [Chi+12] and Orion [DeV+13] generate intrinsics code. PolyMage [MVB15]—a DSL for stencil computations—uses the polyhedral model in order to infer dependencies between different loop iterations and generates annotated, auto-vectorization-friendly C code. See Part II for a thorough discussion of DSLs.

2.3. Whole-Function Vectorization

WFV [KH11; KH12; KKS13; Kar15] multiplies a whole function `L` times and maps each instance to one SIMD lane.[5] Thus, WFV can either be used as a tool to vectorize a loop body or a whole function of data-parallel languages like OpenCL. WFV's vectorization is similar to Sierra's SIMD mode although WFV offers less control over this process: The Sierra programmer can explicitly state the desired data layout and mix several vectorization lengths to a certain extent. Moreover, WFV has not yet been applied in an interprocedural way. The current pragmatic workaround is to transitively inline all function calls. On the other hand, WFV uses a sophisticated data flow analysis, which attempts to find the best possible memory access patterns. Furthermore, WFV supports arbitrarily complex control-flow graphs (CFGs) in contrast to Sierra. Finally, WFV tries to keep control flow that only depends on scalar variables uniform. This can often increase the performance significantly. In Sierra, the programmer has to manually annotate via the types of variables which parts of the CFG

[5] The project has lately been renamed to region vectorizer (RV) and its source code is available as free software. See `https://github.com/cdl-saarland/rv`.

should be vectorized and which can be kept uniform. For these reason, we plan to integrate WFV into Sierra's code generator as future work (see Section 5.2).

Some believed we lacked the programming language to describe your perfect world. But I believe that, as a species, human beings define their reality through suffering and misery.

The Matrix

3 A Quick Tour of Sierra

This chapter informally presents useful SIMD programming idioms that Sierra supports. First, we introduce Sierra's type system. Then, we discuss Sierra's *SIMD mode*.

3.1. Types and Conversions

As already discussed, Sierra introduces a new type constructor: `varying(L)`. Syntactically, this constructor acts as an additional *type qualifier*. The argument `L` must be a *constant expression* [ISO11, §5.19], which must evaluate to a positive integer. Additionally, the type qualifier `uniform` is available, which acts as syntactic sugar for `varying(1)`. This is also the default qualifier if the programmer has not specified one. Thus, `uniform`'s sole purpose is to stress that a variable is scalar.

Applying the `varying` qualifier to an arithmetic type yields a vector of this type. All usual operators are overloaded, so they also work on vectors. However, it is an error to mix vectors with different lengths in an operation.

Example 3.1 (Arithmetic vector types)
The following listing makes use of some arithmetic vectors:

```
short varying(8) s;                    // declaration only
int   varying(4) i = {0, 1, 2, 3};     // initialization list
auto j = (int varying(4)){0, 1, 2, 3};// compound literal
auto k = i + j;                        // int varying(4)

// error: vector lengths mismatch
i + (int varying(2)){0, 1};

// error: vector length not a constant expression
float varying(std::strlen(/*...*/)) x;
```

```
// OK: std::max specified as constexpr
float varying(std::max(2, 4)) y;
```

The C++ standard defines specific rules when and how values from one type are automatically converted to another type. In Sierra these rules apply analogously to vectors:

```
short varying(4) s = /*...*/;
int   varying(4) i = /*...*/;
auto x = s + i; // x is of type int varying(4)
```

3.1.1. Broadcast

Sierra automatically converts a scalar to a vector of the same element type if needed. This is achieved by duplicating the value for each SIMD lane. We call this operation *broadcast*.

Example 3.2 (Broadcast)
Consider the following listing:

```
int uniform    u = /*...*/;
int varying(4) v = /*...*/;
int varying(8) w = /*...*/;
u + v; // u is broadcast to int varying(4)
v + w; // error: vector lenghts mismatch

Vec3 varying(4) cross(Vec3 varying(4) v,
                      Vec3 varying(4) w) { /*...*/ }

Vec3 uniform u;
Vec3 varying(4) v;
Vec3 varying(4) w = cross(u, v); // u is broadcast
```

Moreover, broadcasts and arithmetic conversions may happen in the same expression:

```
short uniform s = /*...*/;
int varying(4) i = /*...*/;
auto x = s + i; // x is of type int varying(4)
```

3.1.2. Pointers and Gather/Scatter

The `density` function in the vectorized volume ray caster (see Section 1.2.2) needs to look up a vector of voxel data from incoherent memory locations. This is expressed via a *gather* from a *vectorial pointer to scalar data* as described in this section.

Both a pointer itself as well as its referenced type may be scalar or vectorial. Hence, four possibilities arise:

scalar pointer to scalar data:

```
int uniform* uniform p;
```

This is an ordinary C++ pointer.

scalar pointer to vectorial data:

```
int varying(4)* uniform p;
```

A scalar pointer to a vector works like a scalar pointer to a scalar: Dereferencing this pointer yields an lvalue of the pointer's referenced type. So, `*p` is of type `int varying(4)`.

vectorial pointer to scalar data:

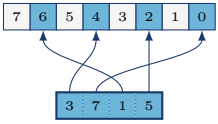
```
int uniform* varying(4) p;
```

Loading from `*p` (called *gather*) has the effect that four *scalars* from different memory locations are loaded. These four scalars are assembled into a new vector. Conversely, assigning a vector `v` to `*p` (called *scatter*) disassembles `v` and writes its components into the designated memory locations. So, `*p` is of type `int varying(4)`.

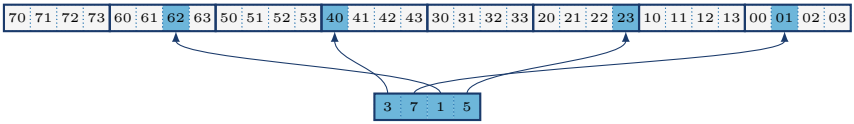
vectorial pointer to vectorial data:

```
int varying(4)* varying(4) p;
```

Loading from `*p` (also called *gather*) has the effect that four *vectors* from different memory locations are loaded. Then, the first element of the first loaded vector, the second element of the second loaded vector, and so forth, are extracted. These four elements are assembled into a new vector. Conversely, assigning a vector `v` to `*p` (also called *scatter*) inserts the first element of `v` into the vector pointed to by the first element of `p`; the second element of `v` is inserted into the vector



(a) a+x is of type `int uniform* varying(4)*` and dereferencing yields {4,0,6,2}



(b) a+x is of type `int varying(4)* varying(4)*` and dereferencing yields {40,01,62,23}

Figure 3.1.: Address computations for vectorial pointers

pointed to by the second element of p, and so forth. So, *p is of type `int varying(4)`.

Analogously, a reference itself and its referenced type may be scalar or vectorial. For example, `int uniform& varying(4)` denotes a vectorial reference to scalar data which implies gathers/scatters.

Furthermore, array indices are allowed to be vectorial in Sierra. Like in C++, an array subscript of the form `E1[E2]` is identical to `(*((E1)+(E2)))` if `E1` is a pointer. However, the semantics of binary + is overloaded to work with vectors. In particular, `E2` will be broadcast to `E1`'s vector length if `E2` is scalar. Similar to arithmetic types, it is an error to mix vectors with different vector lengths.

Example 3.3 (Address computations)
The following code gathers a vector from a *vectorial pointer to scalar data*. See Figure 3.1a for a graphical depiction of the underlying address computation.

```
int uniform a[8] = {7, 6, 5, 4, 3, 2, 1, 0};
int varying(4) x = {3, 7, 1, 5};
int uniform* varying(4)* p = a + x;
int varying(4) v = *p;   // gather: {3, 7, 1, 5}
// same as above using array indexing:
int varying(4) v = a[x]; // gather: {3, 7, 1, 5}
```

The following code gathers a vector from a *vectorial pointer to vectorial data*. See Figure 3.1b for a graphical depiction of the underlying address computation.

```
int varying(4) a[8] = {{70, 71, 72, 73},
                       /*...*/, {00, 01, 02, 03}};
int varying(4) x = {3, 7, 1, 5};
int uniform* varying(4)* p = a + x;
int varying(4) v = *p;   // gather: {3, 7, 1, 5}
// same as above using array indexing:
int varying(4) v = a[x]; // gather: {3, 7, 1, 5}
```

Just like when programming with asynchronous threads, *data races* [ISO11, §1.10] may occur. This will happen if several SIMD lanes want to write to the same memory location.

Example 3.4 (Data races)
Suppose, `p` is a vectorial pointer to a scalar array. The scatter operation in the following listing provokes a data race because both the second and the third SIMD lane want to write to `a + 1`:

```
int uniform a[] = /*...*/;
a[(int varying(4)){0, 1, 1, 3}] = {0, 1, 2, 3};
```

ISPC evaluates the stores of each element in an *unspecified* order; CUDA and OpenCL define these data races as *undefined behavior*; C++ defines multi-threaded data races as *undefined behavior*, too. Sierra needs to decide on one of these strategies. Undefined behavior is arguably the most portable choice.

Example 3.5 (AoS to hybrid SoA conversion)
As discussed in Chapter 1 the hybrid SoA layout is the preferred data layout for SIMD. However, sometimes data is only available in AoS format. The programmer can exploit Sierra's vectorial pointer arithmetic to convert from AoS to AoSoA and vice versa:

```
Vec3 aos[] = /*...*/;

for (int i = 0; i < size; i += L) {
    Vec3 varying(L) v = aos[i + seq<L>()]; // gather
    /* do something with v */
    aos[i + seq<L>()] = v;                 // scatter
}
```

3.1.3. Derived Types

Unlike pointers, **struct**s and **union**s cannot be vectorial. Instead, vectorization is recursively applied to their members. Already specified members remain untouched:

Example 3.6 (Struct)
Using this **struct** declaration

```
struct S {
    int a, b;
    int uniform c;
    int varying(4) d;
};
```

to instantiate different vectors, we obtain the following layouts:

```
//  vector length: #.a #.b #.c #.d
S uniform s;    //  1   1   1   4
S varying(4) t; //  4   4   1   4
S varying(8) u; //  8   8   1   4
```

This schema is recursively applied to all fields. However, when encountering a pointer, the pointer itself becomes vectorial; the referenced type remains untouched. The following example explains the rationale behind this approach.

Example 3.7 (Vectorized linked list)
Using these data types

```
struct ListNode {
    int data;
    ListNode* next;
};

struct List {
    int size;
    ListNode* root;
};
```

we can instantiate a vectorized list with four `size` elements and four `root` pointers to scalar `ListNodes`.

```
List varying(4) vectorized_list;
// ...
ListNode* varying(4) n = vectorized_list->root->next;
int varying(4) data = n->data; // gather
```

If instead the referenced type had been vectorized, we would have gotten four `size` elements and vectorial `ListNodes`. But that makes no sense because we just want to have a single `size` element in this example. On the other hand, the programmer can easily create a list of vectorial data. For instance, vectorial data for STL containers work out-of-the-box: `std::list<int varying(4)>`.

So far we have only considered vectorization of plain-old-data **`struct`**s without any methods. Ideally, the programmer also wants to automatically vectorize full-featured classes when using the **`varying`** type constructer:

```
MyFullFeaturedClass varying(4) x(/*...*/);
if (vector_condition) {
    x.f(/*...*/);
}
```

We believe that automatic vectorization is possible but is subject to certain restrictions. To begin with and as we have already discussed, our current proposal of Sierra is incompatible with certain complex C++ features like exception handling, for instance. While it is possible to define semantics for what it means when some SIMD lanes throw an exception and others do not, the generated code would be very inefficient and inefficient code defeats the whole purpose of SIMD programming. The reason for this inefficiency is that exception handling cannot be mapped to a CFG. Consequently, the code generator cannot vectorize the CFG (see Chapter 5). Furthermore,

vectorization of full-featured classes is an intrusive venture because it virtually interacts with the whole C++ language. For these reasons, we will focus on plain-old-data for the time being. However, classes can be made polymorphic in vector length by using templates.

Example 3.8 (Templated vector length)
The following variant of the `Vec3` class points into the direction of how more sophisticated polymorphic classes can be built (the **`simd`** keyword is described in Section 3.2.1):

```
template<int L>
struct Vec3 {
  simd(L) Vec3(float varying(L) xx,
               float varying(L) yy,
               float varying(L) zz)
    : x(xx), y(yy), z(zz) {}

  simd(L) Vec3<L> operator+(Vec3<L> v) {
    Vec3<L> result;
    result.x = x + v.x;
    result.y = y + v.y;
    result.z = z + v.z;
    return result;
  }

  float varying(L) x, y, z;
};
```

Extract and Insert Elements

Sierra provides the following built-in functions to insert elements into and extract elements from a vector:

```
template<class T, int L>
T extract(const T varying(L)& vec, int i);
template<class T, int L>
void insert(T varying(L)& vec, int i, T val);
```

Non-**`varying`** types like the `Vec3` template class in Example 3.8 can provide their own template specializations for `extract` and `insert`. Thereby, these types integrate with other generic code using `extracts` and `inserts`.

Vector Types vs. Array Types

At first glance, a vector `int varying(L) v` and an array `int a[L]` seem similar. However, there are a number of important differences.

First of all, arrays cannot be passed by value to a function. A function

```
void f(int a[L]) { /*...*/ }
```

is just syntactic sugar for

```
void f(int* a)
```

in C++. The given length `L` has no semantic meaning. Invoking

```
void g(int varying(L) v)
```

on the other hand, really copies the argument to the parameter `v` and the type checker guarantees that `v` has `L` elements.

Moreover, note the difference between

- `vec3 v[N]` (see Figure 1.2a) and
- `vec3 varying(L) v[N]` (see Figure 1.2c).

Furthermore, `vec3 va[N] varying(L)` denotes `L` arrays of scalar data. The type of `&va[23]` is `vec3 uniform* varying(L)`.

In contrast to arrays, vectors do not allow the programmer to take the address of one of its elements. Any attempt to compute the address anyway, e.g. by utilizing `union`s or tricky pointer casts, leads to undefined behavior. This makes it possible for an implementation to choose the exact representation of vectors. For example, the internal representation of a `uint64_t varying(4)` for a machine without native support for vectors of `uint64_t`s may be two vectors of `uint32_t`s: one represents the lower halves, one the upper halves. Moreover, a compiler can optimize more aggressively if it knows that no other part of the program holds a reference to an element of a vector.

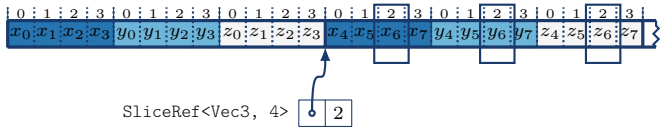
Example 3.9 (No aliasing)
The compiler knows in the following code that `pi` does not alias with any element in `*pv`:

```
void f(int varying(4)* pv, int* pi) { /*...*/ }
```

Programmers can work around this limitation by using a smart reference:

```
template<class T, int L>
class SliceRef {
public:
    SliceRef(T varying(L)& ref, int i)
        : ref_(ref), i_(i) {}

    T get() { return extract(ref_, i); }
    void set(T val) { insert(ref_, i, val); }

private:
    T varying(L)& ref_;
    int i_;
};
```

(a) SliceRef



(b) a SliceRef in action

Figure 3.2.: SliceRef: a smart vectorial reference to an SoA

Example 3.10 (SliceRef)
The data layout in Figure 3.2b arises when a Vec3 **varying**(4) is grouped in an array. If the programmer needs a reference to a logically scalar Vec3, he cannot use a pointer of type Vec3 **uniform*** because the Vec3 instances lie scattered in memory. A SliceRef<Vec3, 4> (see Figure 3.2a) points to the beginning of a Vec3 **varying**(4) and knows the element index (2 in this example) that is referenced.

3.2. SIMD Mode

Please refer to Chapter 1 for a gentle introduction to SIMD mode; Section 4.2 gives a semantic whereas Chapter 5 an implementational view of this execution model. This section highlights how SIMD mode interacts with the rest of C++.

If a control-flow-dependent expression is a vector of length L instead of a scalar, Sierra will enter SIMD mode of length L. All parts of the program depending on that expression are evaluated in that mode. At runtime, each program point must know all active lanes. Therefore, Sierra maintains a value of type `bool varying(L)`, which we call *current mask*. This mask indicates, which SIMD lanes are active. The programmer has read access to this value via the `current_mask` keyword. We call L the *current vector length*. As vector lengths must be constant expressions, the type checker statically knows the vector length of each program point and, consequently, Sierra knows statically which parts must be executed in SIMD mode and in which length.

Sierra does not allow nesting of SIMD modes with different vector lengths, although a reasonable semantics *could* be given. For example, nesting a SIMD mode of length 4 within one of length 2 would effectively result in 8 running threads. We leave research in this area for future work and, currently, the programmer can only declare or use scalars or vectors of length L inside SIMD mode of length L. The size of the element type, however, does not introduce any constraints.

Example 3.11 (Vectors inside SIMD mode)
Since `f < d` causes SIMD mode of length 4, accessing `i` of length 8 is a semantic error:

```
float  varying(4) f = /*...*/;
double varying(4) d = /*...*/;
int    varying(8) i = /*...*/;
if (f < d) {
    d += f; // OK
    i++;    // error
}
```

Scalar variables and control flow on the other hand are always allowed. In fact, keeping control flow scalar whenever possible usually improves performance significantly. The reason for this is that the code generator does not have to linearize control-flow (see Chapter 5) and many scalar operations are more efficient than their vectorial counter parts. In particular, a scalar load/store is vastly more efficient than gather/scatter.

The following statements will trigger SIMD mode of length L for S if Ev is a vector of length L:

- `if (Ev) S [else S]`
- `switch (Ev) S`
- `for (Si; Ev; E) S`
- `while (Ev) S`
- `do S while (Ev);`

Additionally, short-circuit evaluation might trigger SIMD mode of length `L` for `E` if `Ev` is a vector of length `L`:

- `Ev && E` and `E && Ev`
- `Ev || E` and `E || Ev`
- `Ev ? E : E`

The statements `break`, `continue`, `case` and `return` can also be used in SIMD mode. Using these unstructured control-flow statements may cause parts of the program to be executed in SIMD mode, although these parts are not syntactically nested within a vectorial control-flow construct (see Chapter 5). While it is feasible to support arbitrary *goto-statements* and labels, code generation and even type checking becomes complex—in particular when working with a C++-AST (see Section 5.2).

3.2.1. Function Calls

Since arbitrary functions may be called in SIMD mode, these functions need to know about the current mask. For example, the function `density` in Listing 1.4 is called in SIMD mode. Therefore, a function can be annotated with `simd(L)`. This allows Sierra to pass the current mask of length `L` to that function and Sierra will execute that function in SIMD mode of length `L`. If this annotation is missing, the function will only work in scalar mode. Thus, invoking a scalar-mode function in SIMD mode requires Sierra to split currently active vectorial arguments into scalars similar to the `for_each_active` statement (see Section 3.2.3). Calling a `simd(L)` function in scalar mode, requires Sierra to pass an all-`true` mask as current mask to that function.

Example 3.12 (Templated SIMD mode)
This parameter can also be templated.

```
template<int L>
simd(L) float varying(L) dot(Vec3 varying(L) v,
                             Vec3 varying(L) w) {
    return v.x*w.x + v.y*w.y + v.z*w.z;
}
```

3.2.2. The Scalar Statement

Sometimes it is desirable to deactivate vectorization within a function and proceed with scalar computations. Therefore, Sierra offers the **scalar** statement:

```
scalar (m) S
```

This statement saves the current mask in `m`. Then, `S` is executed in scalar mode. Afterwards, the current mask is restored. The `(m)` is optional.

Example 3.13 (Scalar statement)
The following example demonstrates the use of the *scalar-statement*:

```
// scalar mode
int varying(4) a = /*...*/;
int varying(4) b = /*...*/;
if (a < b) {
    // SIMD mode of length 4
    scalar (m) { /*m is of type bool varying(4)*/ }
    // SIMD mode of length 4
}
```

3.2.3. For-Each-Active

A common programming pattern is to fetch all active values from a vector:

```
for_each_active(m, i) S
```

The statement `S` is now in scalar mode and each iteration assigns the next prior active lane index to `i`. The prior active mask is copied to `m`.

```
scalar (m) {
    for (int i=next_true(m, 0); i>=0; i=next_true(m, i+1)) S
}
```

(a) `for_each_active(m, i) S` is equivalent to the above listing

```
for_each_active(m, i) {
    auto x = extract(v, i);
    auto dup = x == v; // indicates duplicates
    // activate duplicates in prior active lanes
    if (m & dup) S
    m &= ~dup;          // blend out duplicates
}
```

(b) `for_each_unique(x, v) S` is equivalent to the above listing

Figure 3.3.: `for_each_active` and `for_each_unique`

Example 3.14 (Scalarize each active lane)
This statement comes in handy if something scalar must be performed in SIMD mode:

```
simd(4) int varying(4) f(int varying(4) v) {
    int varying(4) result;
    for_each_active(m, i)
        insert(res, i, do_sth_scalar(extract(v, i)));
    return result;
}
```

The `for_each_active` statement translates to the code in Figure 3.3a: The *scalar-statement* switches to scalar mode. The `for` loop iterates over all active lanes with the help of the `next_true(m, i)` function. This function essentially performs a bit scan. It returns the index of the next `true` value in `m` greater than or equal to `i`. The function will return `-1` if there is no further `true` value in `m`.

3.2.4. For-Each-Unique

A related idiom scans a vector `v` in each active lane.

```
for_each_unique(x, v) S
```

For each found *unique* value `x`, the body `S` will be executed in SIMD mode. Only lanes that have been active beforehand and contain `x`, will be activated. The iteration order is implementation-defined. The **`for_each_unique`** statement translates to the pattern in Figure 3.3b.[1] The statement's usefulness is best demonstrated by an example.

Example 3.15 (Scalarize each active lane uniquely)
In the following snippet of a rendering code each `Material` instance has a function pointer to compute its transparency. Suppose, the vectorial pointer `m` is `{m1, m2, m2, m1}` and the current mask is `{1, 1, 1, 0}`. Then, **`for_each_unique`** performs two iterations: One iteration invokes `p1->get_transparency` using a current mask of `{1, 0, 0, 0}`; the other iteration invokes `p2->get_transparency` using a current mask of `{0, 1, 1, 0}`.

```
Material uniform* m varying(4) = /*...*/;
Vec3 varying(4) transparency;
for_each_unique(mm, m)
    transparency = mm->get_transparency(mm, /*...*/);
```

Similar code occurs in the OSPRay renderer, which is partially written in ISPC [Wal+17].

Example 3.16 (Virtual method calls)
Similarly, invoking a virtual method on a vectorial **`this`** pointer can be implemented:

```
struct A {
    virtual simd(L) R f(/*params*/) = 0;
};
struct B : public A {
    virtual simd(L) R f(/*params*/) { /*...*/ }
};
struct C : public A {
    virtual simd(L) R f(/*params*/) { /*...*/ }
};
```

[1] Note that only the *scalar-statement* needs to be implemented inside the compiler. Both **`for_each_active`** and **`for_each_unique`** can be implemented as macro. Obviously, error messages can be more precise if these statements are built into the compiler.

```
A* varying(4) a = /*...*/;
a->f(/*args*/);
```

Just like **for_each_unique** Sierra searches the vectorial **this** pointer for duplicates and groups SIMD lanes that reference the same virtual method in one call. The current mask, which Sierra passes to that method, indicates the pointers that were active beforehand and actually reference that method.

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

Benjamin C. Pierce,
Types and Programming Languages

4 Semantics

This chapter takes a formal look on Sierra's type system and semantics. To this end, we introduce Imp—a small imperative language—and show how to extend it with `varying` types and SIMD mode to obtain VecImp. In VecImp the *if-* and *while-statements* are overloaded such that a vector of booleans is also allowed as a controlling condition. Finally, we briefly sketch PolyVecImp, a language that uses type inference in order to infer proper vector types.

The notation used is described in Appendix A; more extensive proofs can be found in Appendix B.

4.1. Imp

Imp mimics a subset of C. The term "subset" is not meant in a strict mathematical sense in this context. It rather relates to the feature set Imp offers compared to C.

On the one hand, Imp behaves more like a real language as opposed to small imperative languages typically formalized in semantics lectures [e.g. Mye13]. This is because Imp supports functions and full recursion. Function calls are particularly interesting when discussing VecImp later on but require a more complex discussion of both Imp and VecImp.

On the other hand, we elide features that are straightforward to add like, for example, `struct` types and appropriate expressions. Furthermore, we would like to study a sound calculus. For this reason, we also elide unsound features like, for instance, C-style (i.e., memory-unsafe) pointers.[1]

[1] see Norrish [Nor98] for a formal discussion of C and undefined behavior

$$
\begin{array}{rcll}
\Gamma & ::= & \varnothing \mid \Gamma, x : t & \text{(typing environment)} \\
\sigma & ::= & \varnothing \mid \sigma, x : v_t & \text{(state)} \\
t & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{float} \mid \bot & \text{(type)} \\
\Phi & ::= & \varnothing \mid \Phi, f & \text{(program)} \\
f & ::= & t\ \ell(\overline{t\ x})\ \{\ s\ \} & \text{(function)} \\
s & ::= & & \text{(statement)} \\
 & & \texttt{skip} & \text{(skip)} \\
 & \mid & \texttt{return}\ e; & \text{(return)} \\
 & \mid & \hat{s}\ s & \text{(list)} \\
\hat{s} & ::= & & \text{(head)} \\
 & & e; & \text{(expression statement)} \\
 & \mid & t\ x; & \text{(declaration)} \\
 & \mid & x = e; & \text{(assignment)} \\
 & \mid & \texttt{if}\ (e)\ s\ \texttt{else}\ s & \text{(if)} \\
 & \mid & \texttt{while}\ (e)\ s & \text{(while)} \\
e & ::= & & \text{(expression)} \\
 & & x & \text{(variable)} \\
 & \mid & \nu_t & \text{(value)} \\
 & \mid & \ell(\overline{e}) & \text{(call)} \\
 & \mid & \{\ s\ \}_\sigma & \text{(statement expression)} \\
a & ::= & s \mid e & \text{(term)}
\end{array}
$$

Figure 4.1.: Syntax of Imp. Expanded syntax is grayed out.

4.1.1. Syntax

Figure 4.1 depicts Imp's syntax. The grayed out part of the syntax is expanded syntax. An Imp program consists of a sequence Φ of functions. Function names range over ℓ, variable names over x. We require all function and variable names to be unique. In order to pass information from one statement to the next one, statements are organized as *lists*, which consist of a *head* $\hat{s}$ and a subsequent statement s. The *skip* and *return* statements mark the end of a statement list. Furthermore, `skip` and `return` ν_t; are *statement values*—they cannot be further reduced. An Imp programmer has access to three types: `bool`, `int` and `float`. We write ν_t to denote

some *expression value* of type t. For example, the value $\nu_{\texttt{bool}}$ is either `true` or `false`.

Term denotes either an expression or a statement. A *term value* is either an expression value or a statement value.

We use the following syntactic sugar in both Imp and VecImp:

$$\begin{aligned} t\ x = e; &\quad\coloneqq\quad t\ x;\ x = e; \\ \texttt{if}\ (e)\ s &\quad\coloneqq\quad \texttt{if}\ (e)\ s\ \texttt{else skip} \\ \texttt{for}\ (t\ x = e_i;\ e_c;\ x_s = e_s)\ s_b &\quad\coloneqq\quad t\ x = e_i;\ \texttt{while}\ (e_c)\ \{\ s_b\ x_s = e_s;\ \} \\ x = e_1 \oplus e_2 &\quad\coloneqq\quad x = \ell_{\oplus,t_1,t_2}(e_1, e_2) \end{aligned}$$

We map binary operators to function calls. We mangle the operator symbol and the expressions' types into a unique function label $\ell_{\oplus,t_1,t_2}$. We assume that the implementation of these functions is built-in.

Additionally, we use $\{\ s\ \}$ in the concrete syntax to disambiguate statement nesting and also elide the closing `skip` within s if applicable. A real language would also "forget" the names declared within the braces. But since we require all names to be unique anyway, we do not model this in our theoretical considerations.

Example 4.1 (Mandelbrot in Imp)
Listing 4.1 computes the famous Mandelbrot set in Imp.

4.1.2. Typing

Imp's type system is presented in Figure 4.2. T-Prg checks all functions in a program. In order to check a function, typing rules maintain a typing environment Γ, which keeps track of each variable and its associated type. T-Fun populates this environment with each parameter and its associated type in order to check a function's body.

Statement typing rules have the form $\Phi; \Gamma \vdash s : t$. Read: Statement s `return`s a value of type t assuming the program Φ and the typing environment Γ. At this point the $\bot$ type, which is part of the expanded syntax, unveils. This type indicates that a statement list ends with `skip`. Note that T-If and T-While allow $\bot$-typed statements as consequence/alternative/body.

Program: $\boxed{\vdash \Phi}$

$$\text{T-Prg}\ \frac{\Phi \vdash f_1 \quad \cdots \quad \Phi \vdash f_n}{\vdash f_1, \ldots, f_n}$$

Function: $\boxed{\Phi \vdash f}$

$$\text{T-Fun}\ \frac{\Phi; \overline{x : t} \vdash s : t_r}{\Phi \vdash t_r\ \ell(\overline{t\ x})\ \{\ s\ \}}$$

Statement: $\boxed{\Phi; \Gamma \vdash s : t}$

$$\text{TS-Skip}\ \frac{}{\Phi; \Gamma \vdash \texttt{skip} : \bot} \qquad \text{TS-Ret}\ \frac{\Phi; \Gamma \vdash e : t}{\Phi; \Gamma \vdash \texttt{return}\ e; : t}$$

$$\text{TS-Expr}\ \frac{\Phi; \Gamma \vdash e : t \qquad \Phi; \Gamma \vdash s : t_r}{\Phi; \Gamma \vdash e;\ s : t_r} \qquad \text{TS-Decl}\ \frac{\Phi; \Gamma[x \mapsto t] \vdash s : t_r}{\Phi; \Gamma \vdash t\ x;\ s : t_r}$$

$$\text{TS-Assign}\ \frac{x : t \in \Gamma \qquad \Phi; \Gamma \vdash e : t \qquad \Phi; \Gamma \vdash s : t_r}{\Phi; \Gamma \vdash x = e;\ s : t_r}$$

$$\text{TS-If}\ \frac{\Phi; \Gamma \vdash e : \texttt{bool} \qquad \Phi; \Gamma \vdash s_t : \bot \qquad \Phi; \Gamma \vdash s_f : \bot \qquad \Phi; \Gamma \vdash s_r : t_r}{\Phi; \Gamma \vdash \texttt{if}\ (e)\ s_t\ \texttt{else}\ s_f\ s_r : t_r}$$

$$\text{TS-While}\ \frac{\Phi; \Gamma \vdash e : \texttt{bool} \qquad \Phi; \Gamma \vdash s_b : \bot \qquad \Phi; \Gamma \vdash s_r : t_r}{\Phi; \Gamma \vdash \texttt{while}\ (e)\ s_b\ s_r : t_r}$$

Expression: $\boxed{\Phi; \Gamma \vdash e : t}$

$$\text{TE-Var}\ \frac{x : t \in \Gamma}{\Phi; \Gamma \vdash x : t} \qquad \text{TE-Val}\ \frac{}{\Phi; \Gamma \vdash \nu_t : t}$$

$$\text{TE-Call}\ \frac{t_r\ \ell(t_1\ x_1, \ldots, t_n\ x_n)\ \{\ s\ \} \in \Phi \qquad \Phi; \Gamma \vdash e_1 : t_1 \qquad \cdots \qquad \Phi; \Gamma \vdash e_n : t_n}{\Phi; \Gamma \vdash \ell(e_1, \ldots, e_n) : t_r}$$

$$\text{TE-Stmt}\ \frac{\Phi; \Gamma \vdash s : t}{\Phi; \Gamma \vdash \{\ s\ \}_\sigma : t}$$

Figure 4.2.: Typing in Imp

Evaluation context:

$$\mathcal{E}[\star] \coloneqq \texttt{return}\ \star; \mid \star;\ s \mid x = \star;\ s \mid \texttt{if}\ (\star)\ s\ \texttt{else}\ s\ s \mid\ \ell(\overline{\nu_t}, \star, \overline{e}) \mid \{\ \star\ \}_{\hat{\sigma}}$$

$$\text{E-Eval}\ \frac{\Phi \vdash \sigma; a \to \sigma'; a'}{\Phi \vdash \sigma; \mathcal{E}[a] \to \sigma'; \mathcal{E}[a']}$$

Statement: $\boxed{\Phi \vdash \sigma; s \to \sigma'; s'}$

$$\text{ES-Expr}\ \frac{}{\Phi \vdash \sigma; \nu_t;\ s \to \sigma; s} \qquad \text{ES-Decl}\ \frac{}{\Phi \vdash \sigma; t\ x;\ s \to \sigma[x \mapsto 0_t]; s}$$

$$\text{ES-Assign}\ \frac{x : \nu'_t \in \sigma}{\Phi \vdash \sigma; x = \nu_t;\ s \to \sigma[x \mapsto \nu_t]; s}$$

$$\text{ES-IfT}\ \frac{}{\Phi \vdash \sigma; \texttt{if}\ (\texttt{true})\ s_t\ \texttt{else}\ s_f\ s_r \to \sigma; s_t \circ s_r}$$

$$\text{ES-IfF}\ \frac{}{\Phi \vdash \sigma; \texttt{if}\ (\texttt{false})\ s_t\ \texttt{else}\ s_f\ s_r \to \sigma; s_f \circ s_r}$$

$$\text{ES-While}\ \frac{}{\Phi \vdash \sigma; \texttt{while}\ (e)\ s_b\ s_r \to \sigma; \texttt{if}\ (e)\ \{\ s_b\ \texttt{while}\ (e)\ s_b\ \}\ s_r}$$

Expression: $\boxed{\Phi \vdash \sigma; e \to \sigma'; e'}$

$$\text{EE-Var}\ \frac{x : \nu_t \in \sigma}{\Phi \vdash \sigma; x \to \sigma; \nu_t} \qquad \text{EE-Call}\ \frac{t_r\ \ell(\overline{t\ x})\ \{\ s\ \} \in \Phi}{\Phi \vdash \sigma; \ell(\overline{\nu_t}) \to \overline{x : \nu_t}; \{\ s\ \}_\sigma}$$

$$\text{EE-Stmt}\ \frac{}{\Phi \vdash \sigma; \{\ \texttt{return}\ \nu_t;\ \}_{\sigma'} \to \sigma'; \nu_t}$$

Figure 4.3.: Evaluation in IMP

```
int
mandel(float x0, float y0,
       float x1, float y1) {
    float dx = (x1 - x0) / WIDTH;
    float dy = (y1 - y0) / HEIGHT;
    for (int j = 0; j < HEIGHT; j = j + 1) {
        for (int i = 0; i < WIDTH; i = i + 1) {

            float x = x0 + i*dx;
            float y = y0 + j*dy;
            store(iter(x, y), i, j);
        }
    }
    return 0;
}

int iter(float cr,
         float ci) {
    float zr = cr;
    float zi = ci;
    int i = 0;
    while ((i < MAX_ITER) & (zr*zr + zi*zi < 4.f)) {
        float newr = zr*zr - zi*zi;
        float newi = 2.f*zr*zi;
        zr = cr + newr;
        zi = ci + newi;
        i = i + 1;
    }
    return i;
}
```

Listing 4.1.: Mandelbrot set computation in Imp

Expression typing rules have the form $\Phi; \Gamma \vdash e : t$. Read: Expression e is of type t assuming the program Φ and the typing environment Γ.

> *Remark.* Statement and expression typing rules are two relations defined in a mutual recursive way.

```
simd(1) int varying(1)
mandel(float varying(1) x0, float varying(1) y0,
       float varying(1) x1, float varying(1) y1) {
    float varying(1) dx = (x1 - x0) / WIDTH;
    float varying(1) dy = (y1 - y0) / HEIGHT;
    for (int j = 0; j < HEIGHT; j = j + 1) {
        for (int varying(1) ii = 0; ii < WIDTH; ii = ii + 4) {
            int varying(4) i = ii + {0, 1, 2, 3};
            float varying(4) x = x0 + i*dx;
            float varying(4) y = y0 + j*dy;
            store(iter(x, y), i, j);
        }
    }
    return 0;
}

simd(1) int varying(4) iter(float varying(4) cr,
                            float varying(4) ci) {
    float varying(4) zr = cr;
    float varying(4) zi = ci;
    int varying(4) i = 0
    while ((i < MAX_ITER) & (zr*zr + zi*zi < 4.f)) {
        float varying(4) newr = zr*zr - zi*zi;
        float varying(4) newi = 2.f*zr*zi;
        zr = cr + newr;
        zi = ci + newi;
        i = i + 1;
    }
    return i;
}
```

Listing 4.2.: Mandelbrot set computation in VECIMP

4.1.3. Evaluation

Dynamic semantics is presented by a small step semantics (see Figure 4.3). During evaluation the state σ keeps track of each variable and its associated value. Statement rules have the form $\Phi \vdash \sigma; s \to \sigma'; s'$. Read: Statement s in state σ evaluates in one step to statement s' in state σ' assuming the program Φ. Similarly, expression rules have the form $\Phi \vdash \sigma; e \to \sigma'; e'$.

The typing environment is an overapproximation of the state as the following definition and lemma summarize.

Definition 4.1 (Typing Environment and State Agreement)
We define:

$$\Gamma \trianglerighteq \sigma \quad :\Leftrightarrow \quad dom(\Gamma) = dom(\sigma) \quad \wedge \quad \forall (x : \nu_t) \in \sigma : \quad (x : t) \in \Gamma \ .$$

Lemma 4.1 (Uniqueness)
If $\Gamma_1 \trianglerighteq \sigma$ *and* $\Gamma_2 \trianglerighteq \sigma$, *then* $\Gamma_1 = \Gamma_2$.

Proof. By a straightforward induction. □

Rule E-Eval uses an evaluation context $\mathcal{E}[\star]$ in order to instantiate rules for a whole family of terms. Note that $\mathcal{E}$'s hole $\star$ may either be a statement or expression. Thus, the resulting relation is either a statement or expression relation. This rule evaluates all subterms to term values in a deterministic order. All other evaluation rules deal with the case how a term evaluates if all other subterms are values.

ES-IfT and ES-IfF concatenate two statements s_1 and s_2 where s_1 must end with `skip`. This prerequisite is checked by T-If and T-While.

$$s_1 \circ s_2 \coloneqq \hat{s}_1 \ s_2 \quad \text{where } s_1 = \hat{s}_1 \ \texttt{skip}$$

Lemma 4.2 (Statement Sequence)

$$\textit{If} \quad \Phi;\Gamma \vdash s_1 : \bot \quad \textit{and} \quad \Phi;\Gamma \vdash s_2 : t_2 \ , \quad \textit{then} \quad \Phi;\Gamma \vdash s_1 \circ s_2 : t_2 \ .$$

Proof. By a straightforward induction. □

In order to support recursive function calls, EE-Call must store the current state before descending into the called function. This is the occasion where the expanded syntax of a *statement expression* (not to be confused with an *expression statement*) is needed:

- The call evaluates to a statement expression $\{\ s\ \}_\sigma$.
- The statement s is the called function's body.
- The statement expression memorizes the current state σ.
- EE-Call constructs a new state, which only consists of all parameters bound to the arguments' values. In particular, all prior variable bindings are now "forgotten".

In order to return from that call, EE-Stmt evaluates $\{\ \texttt{return}\ \nu_t;\ \}_{\sigma'}$ to ν_t while restoring the previously memorized state σ'. By doing this, EE-Stmt discards the previous state σ. Statement expressions are inspired by a GCC C-extension.[2]

> *Remark.* Similar to the typing rules, statement and expression evaluation rules are defined in a mutual recursive way.

4.1.4. Soundness

Lemma 4.3 (IMP: Progress)
Every IMP *term is either a term value or can be stepped into another term. To be more precise: Let* $\vdash \Phi$ *and* $\Gamma \trianglelefteq \sigma$.

$$\textit{If}\quad \begin{array}{l}\Phi;\Gamma \vdash s : t\\ \Phi;\Gamma \vdash e : t\end{array},\quad \textit{then}\quad \begin{array}{l} s = \texttt{skip} \ \vee\ s = \texttt{return}\ \nu_t;\\ e = \nu_t\end{array}\quad \textit{or}\quad \begin{array}{l}\exists s', \sigma' : \Phi \vdash \sigma; s \to \sigma'; s'\\ \exists e', \sigma' : \Phi \vdash \sigma; e \to \sigma'; e'\end{array}.$$

Proof sketch. By mutual induction on a derivation of $\begin{array}{l}\Phi;\Gamma \vdash s : t\\ \Phi;\Gamma \vdash e : t\end{array}$.
See Section B.1 for the full proof.

Lemma 4.4 (IMP: Preservation)
If a well-typed IMP *term takes a step of evaluation, the resulting term is also well-typed. To be more precise: Let* $\vdash \Phi$, $\Gamma \trianglelefteq \sigma$, *and* $\Gamma' \trianglelefteq \sigma'$.

$$\textit{If}\quad \begin{array}{l}\Phi;\Gamma \vdash s : t\\ \Phi;\Gamma \vdash e : t\end{array}\quad \textit{and}\quad \begin{array}{l}\Phi \vdash \sigma; s \to \sigma'; s'\\ \Phi \vdash \sigma; e \to \sigma'; e'\end{array},\quad \textit{then}\quad \begin{array}{l}\Phi;\Gamma' \vdash s' : t\\ \Phi;\Gamma' \vdash e' : t\end{array}.$$

Proof sketch. By mutual induction on a derivation of $\begin{array}{l}\Phi \vdash \sigma; s \to \sigma'; s'\\ \Phi \vdash \sigma; e \to \sigma'; e'\end{array}$.
See Section B.1 for the full proof.

[2] see https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html

Γ	::= as in Imp	(typing environment)
σ	::= as in Imp	(state)
m	::= $\nu_{\texttt{bool varying}(l)}$	(current mask)
τ	::= `bool` | `int` | `float`	(element type)
t	::= τ `varying`(l) | $\bot$	(type)
l	::= $1 \mid 2 \mid 3 \mid 4 \mid \cdots$	(vector length)
Φ	::= as in Imp	(program)
f	::= `simd`(l) t $\ell(\overline{t\ x})$ { s }	(function)
s	::= as in Imp	(statement)
$\hat{s}$	::= as in Imp	(head)
e	::= as in Imp	(expression)
	| $\{\overline{e}\}$	(vector)
a	::= as in Imp	(term)

Figure 4.4.: Syntax of VecImp. Expanded syntax is grayed out.

Theorem 4.1 (Imp: Soundness)
No well-typed Imp *term gets stuck. To be more precise: Let* $\vdash \Phi$, $\Gamma \trianglerighteq \sigma$, *and* $\Gamma' \trianglerighteq \sigma'$.

$$\textit{If} \quad \begin{array}{l}\Phi;\Gamma \vdash s : t \\ \Phi;\Gamma \vdash e : t\end{array} \quad \textit{and} \quad \begin{array}{l}\Phi;\sigma;s \to^* \sigma';s' \\ \Phi;\sigma;e \to^* \sigma';e'\end{array},$$

$$\textit{then} \quad \begin{array}{l}s' = \texttt{skip} \\ \\ e' = \nu_t\end{array} \quad \textit{or} \quad s' = \texttt{return}\ \nu_t; \quad \textit{or} \quad \begin{array}{l}\exists s'', \sigma'' : \Phi;\sigma';s' \to \sigma'';s'' \\ \exists e'', \sigma'' : \Phi;\sigma';e' \to \sigma'';e''\end{array}.$$

Proof. By Lemma 4.3 and Lemma 4.4. □

4.2. VecImp

This section discusses how to extend Imp to its vector counterpart VecImp by adding vector types and SIMD mode.

4.2.1. Syntax

Figure 4.4 depicts the required modifications of Imp's syntax in order to obtain VecImp. In contrast to Imp, each function in VecImp must

be annotated with a *vector length* l (see Section 3.2.1). Furthermore, in VecImp all types accessible to the programmer must be qualified with a vector length. The following function retrieves the vector length of a type:

$$\begin{aligned} \|\tau \texttt{ varying}(l)\| &= l \\ \|\bot\| &= 1 \end{aligned}$$

Finally, VecImp adds a *vector expression* in order to construct a vector. For example, assuming `x` is of type `int`, the expression `{0, 1, x, 3}` is of type `int varying(4)` (see TE-Vec). Like in Imp, ν_t denotes some *expression value* of type t. In particular, `{0, 1, 2, 3}` is an example for a value $\nu_{\texttt{int varying(4)}}$ of type `int varying(4)`.

> **Example 4.2** (Mandelbrot in VecImp)
> Listing 4.2 computes the Mandelbrot set in VecImp. Note that all types must include the `varying` qualifier. This example uses a vectorization length of 4. The inner loop beginning in Listing 7 vectorizes in x-direction whereas the setup of the outer loop is still scalar. The inner loop assumes that `WIDTH` is a multiple of 4 and computes 4 iterations in lockstep. Note that the condition of the `while` loop in line 22 is of type `bool varying(4)`. This causes the corresponding body to be *vectorized*.

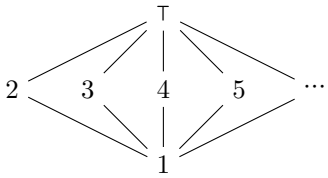
4.2.2. Typing

VecImp checks each function in a program—just like Imp (see T-Prg in Figure 4.2). As soon as VecImp checks a function, VecImp has to keep track of the *current vector length* (see Figure 4.6). For this reason, VecImp adds the current vector length l to VecImp's statement and expression typing relations. If $l \neq 1$, the program is in SIMD mode.

Combining Vector Lengths

The current vector length restricts the occurrence of variables. For example, TE-Var guarantees that the accessed variable

- is scalar,
- occurs in scalar mode or



(a) Hasse diagram of the lattice for $l_d = l \sqcup l_s$

Assignable: $\boxed{l \vdash l \leftarrow l}$

$$\text{A-Length}\ \frac{l_d = l \sqcup l_s}{l \vdash l_d \leftarrow l_s} \qquad \text{A-Splat}\ \frac{}{1 \vdash l \leftarrow 1} \qquad \text{A-Scalar}\ \frac{}{l \vdash 1 \leftarrow 1}$$

(b) assignable relation

Figure 4.5.: Length and assignable relations

- is compatible with the current vector length.

In order to track those restrictions, we use the lattice depicted in Figure 4.5a. This lattice captures the intuition that everything is still well-typed as long as scalar values are used. As soon as a vector comes into play, we ascend in the lattice to—let us say vector length 4. At that point it is still fine to use scalar computations and other vectorial computations of vector length 4. But if we mix those computations with a vector of a different vector length—let us say 8—we ascend in the lattice to $\top$, which indicates a type error. However, we want to rule out type errors in the first place in VecImp's typing rules; we do not want to add many premises of the form "$l_d \neq \top$". For this reason, *we implicitly assume premises of the form $l_d \neq \top$ for all $\ell_d = \ell \sqcup \ell_s$ that occur in the rules.*

Example 4.3 (Length)
It holds $1 = 1 \sqcup 1$, $4 = 1 \sqcup 4$, and $\top = 4 \sqcup 8$. Thus, we cannot conclude:

$$\text{TE-Val}\ \frac{\overbrace{\top = 4 \sqcup 2}^{\lightning}}{\varnothing; \varnothing; 4 \vdash \nu_{\texttt{int varying(2)}} : \texttt{int varying}(2)}\ \lightning \quad .$$

```
int4 v = {0, 8, 7, 1};
bool4 mask = lt4(v, broadcast4(3));
v = blend4(     mask,  add4(v, broadcast4(2)), v);
v = blend4(neg4(mask), sub4(v, broadcast4(3)), v);
print(v);
```

Listing 4.3.: Straightforward implementation of Listing 1.1

Assignments

Both TS-Assign and TE-Call have a notion of assignment. This logic is factored out in the *assignable* relation in Figure 4.5b. A-Length checks that the vector length l_d of the destination variable is the join of the vector length l_s of the source expression and the current vector length l. Moreover, A-Scalar allows assignments from a scalar expression to a scalar variable in SIMD mode, whereas A-Splat allows a broadcast in scalar mode. Note that A-Length already covers a broadcast to a vector in SIMD mode.

4.2.3. Evaluation

Evaluation (see Figure 4.7) must keep track of which vector lanes are active—the *current mask* (see Section 3.2). For this reason, VecImp maintains a boolean vector m. It is important for VecImp's soundness that $\|m\|$ is either scalar or equal to l. The former case means that the current mask is broadcastable to the current vector length.

Definition 4.2 (Vector Length Agreement)
We define:

$$l \trianglerighteq m \quad :\Leftrightarrow \quad \|m\| = 1 \vee l = \|m\|$$

In order to better understand the semantics of SIMD mode, it is a good idea to recap Example 1.2. However, Listing 4.3 is a more straightforward implementation of Listing 1.1 that better reflects the semantics in Figure 4.7.

When evaluating an *if-statement*, E-Eval ensures that the controlling condition evaluates to a value just like in Imp. But in contrast to Imp, VecImp cannot just select the consequence or alternative depending on the condition because the condition can be vectorial. For this reason, VecImp linearizes the control flow by firstly evaluating the consequence (see ES-IfT) and then the alternative (see ES-IfF) with an appropriate mask. Note how

Function: $\boxed{\Phi \vdash f}$

$$\text{T-Fun}\ \frac{\_ = l \sqcup \|t_1\| \quad \cdots \quad \_ = l \sqcup \|t_n\| \quad \_ = l \sqcup \|t_r\| \quad \Phi; \overline{x : t}; l \vdash s : t_r}{\Phi \vdash \texttt{simd}(l)\ t_r\ \ell(t_1\ x_1, \ldots, t_n\ x_n)\ \{\ s\ \}}$$

Statement: $\boxed{\Phi; \Gamma; l \vdash s : t}$

$$\text{TS-Skip}\ \frac{}{\Phi; \Gamma; l \vdash \texttt{skip} : \bot} \qquad \text{TS-Ret}\ \frac{\Phi; \Gamma; l \vdash e : \tau\ \texttt{varying}(l_e) \quad l_r = l \sqcup l_e}{\Phi; \Gamma; l \vdash \texttt{return}\ e; : \tau\ \texttt{varying}(l_r)}$$

$$\text{TS-Expr}\ \frac{\Phi; \Gamma; l \vdash e : t \quad \_ = l \sqcup \|t\| \quad \Phi; \Gamma; l \vdash s : t_r}{\Phi; \Gamma; l \vdash e;\ s : t_r}$$

$$\text{TS-Decl}\ \frac{\Phi; \Gamma[x \mapsto t]; l \vdash s : t_r \quad \_ = l \sqcup \|t\|}{\Phi; \Gamma; l \vdash t\ x;\ s : t_r}$$

$$\text{TS-Assign}\ \frac{x : \tau\ \texttt{varying}(l_x) \in \Gamma \quad \Phi; \Gamma; l \vdash e : \tau\ \texttt{varying}(l_e) \quad l \vdash l_x \leftarrow l_e \quad \Phi; \Gamma; l \vdash s : t_r}{\Phi; \Gamma; l \vdash x = e;\ s : t_r}$$

$$\text{TS-If}\ \frac{\Phi; \Gamma; l \vdash e : \texttt{bool varying}(l_c) \quad l' = l \sqcup l_c \quad \Phi; \Gamma; l' \vdash s_t : \bot \quad \Phi; \Gamma; l' \vdash s_f : \bot \quad \Phi; \Gamma; l \vdash s_r : t_r}{\Phi; \Gamma; l \vdash \texttt{if}\ (e)\ s_t\ \texttt{else}\ s_f\ s_r : t_r}$$

$$\text{TS-While}\ \frac{\Phi; \Gamma; l \vdash e : \texttt{bool varying}(l_c) \quad l' = l \sqcup l_c \quad \Phi; \Gamma; l' \vdash s_b : \bot \quad \Phi; \Gamma; l \vdash s_r : t_r}{\Phi; \Gamma; l \vdash \texttt{while}\ (e)\ s_b\ s_r : t_r}$$

Expression: $\boxed{\Phi; \Gamma; l \vdash e : t}$

$$\text{TE-Var}\ \frac{x : t \in \Gamma \quad \_ = l \sqcup \|t\|}{\Phi; \Gamma; l \vdash x : t} \qquad \text{TE-Val}\ \frac{\_ = l \sqcup \|t\|}{\Phi; \Gamma; l \vdash \nu_t : t}$$

$$\text{TE-Call}\ \frac{\begin{array}{c}\texttt{simd}(l_\ell)\ t_r\ \ell(t_{x_1}\ x_1, \ldots, t_{x_n}\ x_n)\ \{\ s\ \} \in \Phi \\ \Phi; \Gamma; l \vdash e_1 : t_1 \quad \cdots \quad \Phi; \Gamma; l \vdash e_n : t_n \\ l \vdash \|t_{x_1}\| \leftarrow \|t_1\| \quad \cdots \quad l \vdash \|t_{x_n}\| \leftarrow \|t_n\| \quad l \vdash l_\ell \leftarrow l \quad \_ = l \sqcup \|t_r\|\end{array}}{\Phi; \Gamma; l \vdash \ell(e_1, \ldots, e_n) : t_r}$$

$$\text{TE-Stmt}\ \frac{\Phi; \Gamma; l \vdash s : t \quad \_ = l \sqcup \|t\|}{\Phi; \Gamma; l \vdash \{\ s\ \}_\sigma : t}$$

$$\text{TE-Vec}\ \frac{\Phi; \Gamma; l \vdash e_1 : \tau\ \texttt{varying}(1) \quad \cdots \quad \Phi; \Gamma; l \vdash e_n : \tau\ \texttt{varying}(1) \quad \_ = l \sqcup n}{\Phi; \Gamma; l \vdash \{e_1, \ldots, e_n\} : \tau\ \texttt{varying}(n)}$$

Figure 4.6.: Typing in VecImp

Evaluation context:

$$\mathcal{E}[\star] \coloneqq \text{as in Imp} \mid \{\overline{\nu_t}, \star, \overline{e_t}\}$$

$$\text{E-Eval}\ \frac{\Phi; m \vdash \sigma; a \to \sigma'; a'}{\Phi; m \vdash \sigma; \mathcal{E}[a] \to \sigma'; \mathcal{E}[a']}$$

Statement: $\boxed{\Phi; m \vdash \sigma; s \to \sigma'; s'}$

$$\text{ES-Expr}\ \frac{}{\Phi; m \vdash \sigma; \nu_t;\ s \to \sigma; s} \qquad \text{ES-Decl}\ \frac{}{\Phi; m \vdash \sigma; t\ x;\ s \to \sigma[x \mapsto 0_t]; s}$$

$$\text{ES-Assign}\ \frac{x : \nu'_{\tau\ \texttt{varying}(l_x)} \in \sigma \qquad \text{blend}(m, \nu_{\tau\ \texttt{varying}(l_e)}, \nu'_{\tau\ \texttt{varying}(l_x)})\ \text{valid}}{\Phi; m \vdash \sigma; x = \nu_{\tau\ \texttt{varying}(l_e)};\ s \to \sigma[x \mapsto \text{blend}(m, \nu_{\tau\ \texttt{varying}(l_e)}, \nu'_{\tau\ \texttt{varying}(l_x)})]; s}$$

$$\text{ES-IfT}\ \frac{m \wedge \nu_{\texttt{bool varying}(l)}\ \text{valid} \qquad \Phi; m \wedge \nu_{\texttt{bool varying}(l)} \vdash \sigma; s_t \to \sigma'; s'_t}{\Phi; m \vdash \sigma; \texttt{if}\ (\nu_{\texttt{bool varying}(l)})\ s_t\ \texttt{else}\ s_f\ s_r \to \sigma'; \texttt{if}\ (\nu_{\texttt{bool varying}(l)})\ s'_t\ \texttt{else}\ s_f\ s_r}$$

$$\text{ES-IffF}\ \frac{m \wedge \neg\nu_{\texttt{bool varying}(l)}\ \text{valid} \qquad \Phi; m \wedge \neg\nu_{\texttt{bool varying}(l)} \vdash \sigma; s_f \to \sigma'; s'_f}{\begin{array}{l}\Phi; m \vdash \sigma; \texttt{if}\ (\nu_{\texttt{bool varying}(l)})\ \texttt{skip else}\ s_f\ s_r \\ \to \sigma'; \texttt{if}\ (\nu_{\texttt{bool varying}(l)})\ \texttt{skip else}\ s'_f\ s_r\end{array}}$$

$$\text{ES-If}\ \frac{}{\Phi; m \vdash \sigma; \texttt{if}\ (\nu_{\texttt{bool varying}(l)})\ \texttt{skip else skip}\ s_r \to \sigma; s_r}$$

$$\text{ES-While}\ \frac{}{\Phi; m \vdash \sigma; \texttt{while}\ (e)\ s_b\ s_r \to \sigma; \texttt{if}\ (e)\ \{\ s_b\ \texttt{while}\ (e)\ s_b\ \}\ s_r}$$

Expression: $\boxed{\Phi; m \vdash \sigma; e \to \sigma'; e'}$

$$\text{EE-Var}\ \frac{x : \nu_t \in \sigma}{\Phi; m \vdash \sigma; x \to \sigma; \nu_t} \qquad \text{EE-Call}\ \frac{\texttt{simd}(l)\ t_r\ \ell(\overline{t\ x})\ \{\ s\ \} \in \Phi}{\Phi; m \vdash \sigma; \ell(\overline{\nu_t}) \to \overline{x : \nu_t}; \{\ s\ \}_\sigma}$$

$$\text{EE-Stmt}\ \frac{}{\Phi; m \vdash \sigma; \{\ \texttt{return}\ \nu_t;\ \}_{\sigma'} \to \sigma'; \nu_t}$$

Figure 4.7.: Evaluation in VecImp

ES-Assign uses the *blend operation* $\text{blend}(m, b, a)$ to update only active lanes. As indicated by $\|m\| \vdash \|a\| \leftarrow \|b\|$, broadcasts may be necessary (see Section 3.1.1). Note that $\text{blend}(m, b, a)$ is valid iff $\|m\| \vdash \|a\| \leftarrow \|b\|$. ES-IfT and ES-IfF perform calculations on the mask:

- $\neg m$ negates all elements in m;
- $m \wedge m'$ performs a bitwise AND in an element-wise manner. Additionally, m or m' may be scalars. In this case, m/m' is broadcast to the other operand's vector length if applicable.

Note that $m \wedge m'$ is valid iff $\_ = \|m\| \sqcup \|m'\|$. Furthermore, if $m_r = m \wedge m'$, then $\|m_r\| = \|m\| \sqcup \|m'\|$. Finally, if both the consequence and the alternative have been evaluated to `skip`, evaluation continues with the rest of the program in s_r (see ES-If).

4.2.4. Soundness

Lemma 4.5 (VecImp: Progress)
Every VecImp *term is either a term value or can be stepped into another term. To be more precise: Let* $\vdash \Phi$, $\Gamma \trianglerighteq \sigma$, *and* $l \trianglerighteq m$.

$$\textit{If}\quad \begin{array}{l}\Phi;\Gamma;l \vdash s : t\\ \Phi;\Gamma;l \vdash e : t\end{array},$$

$$\textit{then}\quad \begin{array}{l} s = \texttt{skip}; \\ \quad\vee\ e = \nu_t \end{array} \quad \textit{or} \quad \begin{array}{l}\exists s', \sigma' : \Phi; m \vdash \sigma; s \to \sigma'; s'\\ \exists e', \sigma' : \Phi; m \vdash \sigma; e \to \sigma'; e'\end{array}.$$

Proof sketch. By mutual induction on a derivation of $\begin{array}{l}\Phi;\Gamma;l \vdash s : t\\ \Phi;\Gamma;l \vdash e : t\end{array}$.
See Section B.2 for the full proof.

Lemma 4.6 (VecImp: Preservation)
If a well-typed VecImp *term takes a step of evaluation, the resulting term is also well-typed. To be more precise: Let* $\vdash \Phi$, $\Gamma \trianglerighteq \sigma$, $\Gamma' \trianglerighteq \sigma'$, *and* $l \trianglerighteq m$.

$$\textit{If}\quad \begin{array}{l}\Phi;\Gamma;l \vdash s : t\\ \Phi;\Gamma;l \vdash e : t\end{array} \quad \textit{and} \quad \begin{array}{l}\Phi;m \vdash \sigma;s \to \sigma';s'\\ \Phi;m \vdash \sigma;e \to \sigma';e'\end{array}, \quad \textit{then} \quad \begin{array}{l}\Phi;\Gamma';l \vdash s' : t\\ \Phi;\Gamma';l \vdash e' : t\end{array}.$$

Proof sketch. By mutual induction on a derivation of $\begin{array}{l}\Phi;m \vdash \sigma;s \to \sigma';s'\\ \Phi;m \vdash \sigma;e \to \sigma';e'\end{array}$.
See Section B.2 for the full proof.

```
int mandel(float x0, float y0, float x1, float y1) {
    float dx = (x1 - x0) / WIDTH;
    float dy = (y1 - y0) / HEIGHT;
    for (int j = 0; j < HEIGHT; j = j + 1) {
        for (int ii = 0; ii < WIDTH; ii = ii + 4) {
            int i = ii + {0, 1, 2, 3};
            float x = x0 + i*dx;
            float y = y0 + j*dy;
            store(iter(x, y), i, j);
        }
    }
    return 0;
}
```

Listing 4.4.: Mandelbrot set computation in PolyVecImp; `iter` is identical to the Imp version (see Listing 4.1).

Theorem 4.2 (VecImp: Soundness)
No well-typed VecImp *term gets stuck. To be more precise: Let* $\vdash \Phi$, $\Gamma \trianglerighteq \sigma$, $\Gamma' \trianglerighteq \sigma'$, *and* $l \trianglerighteq \|m\|$.

$$\textit{If} \quad \begin{array}{l} \Phi;\Gamma,l \vdash s : t \\ \Phi;\Gamma,l \vdash e : t \end{array} \quad \textit{and} \quad \begin{array}{l} \Phi;m \vdash \sigma;s \to^* \sigma';s' \\ \Phi;m \vdash \sigma;e \to^* \sigma';e' \end{array},$$

$$\textit{then} \quad \begin{array}{l} s' = \texttt{skip}; \\ \quad \vee \; s' = \texttt{return}\ \nu_t; \\ \quad\quad e' = \nu_t \end{array} \quad \textit{or} \quad \begin{array}{l} \exists s'', \sigma'' : \Phi; m \vdash \sigma', s' \to \sigma''; s'' \\ \exists e'', \sigma'' : \Phi; m \vdash \sigma', e' \to \sigma''; e'' \end{array}.$$

Proof. By Lemma 4.5 and Lemma 4.6. □

4.3. PolyVecImp

We have often stressed the similarities between the scalar and the vectorial version of a program. This raises the question whether appropriate `varying` annotations can be inferred. The answer is "yes".

This section briefly presents yet another language called PolyVecImp, which uses type inference in order to propagate vector lengths. To our knowledge, there is no other language that supports this kind of polymorphism. But even this section only provides a brief sketch. We leave a thorough discussion of this topic as future work.

PolyVecImp differentiates between *mono* and *poly types*. This is a shorthand for *monomorphic* and *polymorphic*, respectively, and should not be confused with `uniform` and `varying`. Poly types are types *without* a specified vector length like `int` or `float`; mono types are types *with* an explicitly specified vector length like `int varying(1)` or `float varying(8)`:

$$
\begin{aligned}
f &::= \text{as in Imp} && \text{(function)}\\
\tau &::= \texttt{bool} \mid \texttt{int} \mid \texttt{float} && \text{(element type)}\\
t &::= \tau && \text{(poly type)}\\
&\mid\ \tau\ \texttt{varying}(l) && \text{(mono type)}\\
&\cdots\ \text{rest as in VecImp}
\end{aligned}
$$

In contrast to VecImp, functions are not annotated with `simd`(l). Instead, all functions are polymorphic in their vector length; moreover, all functions are polymorphic in the vector lengths of all poly-typed parameters. The type checker optimistically assumes that all poly types are scalar. According to the lattice in Figure 4.5a and rules similar to Figure 4.6, the type checker corrects vector lengths. When inspecting a function call, PolyVecImp tries to instantiate a function with the current inferred vector length and the vector lengths of the inferred argument types. If this fails, PolyVecImp ascends in the lattice where necessary and retries. Since the call graph may be arbitrarily complex and even cyclic, PolyVecImp must use a fixed-point iteration. If no valid configuration can be inferred, the program is ill-typed.

Example 4.4 (Min in PolyVecImp)
Consider the following function:

```
int min(int a, int b) {
    int result = a;
    if (b < a)
        result = b;
    return result;
}
```

Suppose a call site `min(u, v)` in scalar mode where `u` has mono type `int varying(1)` and `v` has mono type `int varying(4)`. For this specialization of `min` it seems at first glance that PolyVecImp infers the mono type `int varying(1)` for the poly-typed local variable `result` due to the initialization of this variable with `a`. However, the assignment with `b` yields

mono type `int varying(4)` for `result` in this specialization. Hence, the function's return type is `int varying(4)` for the given call site.

At some point the programmer must tell the compiler when and how to initiate vectorization by using mono types. The vector lengths of all other types can be inferred using poly types.

Example 4.5 (Mandelbrot in PolyVecImp)
Listing 4.4 computes the Mandelbrot set in PolyVecImp. The setup of the loop nest itself is similar to VecImp although no `varying` annotations are needed in PolyVecImp because the expression `{0, 1, 2, 3}` is of type `int varying(4)`. This causes `i` to have the same type while `x` is of type `float varying(4)`. Thus, `iter` is instantiated with `float varying(4)` and `float varying(1)` in scalar mode. With this configuration PolyVecImp infers that all local variables must be vectors of length 4. Note that `iter` is syntactically identical to the Imp version.

Once PolyVecImp has found a valid configuration, it is straightforward to translate a PolyVecImp program with the help of the inferred vector lengths to VecImp.

On the other side of the screen,
it all looks so easy.

Tron

5 Code Generation

This chapter discusses how to generate code for VecImp. We assume that the VecImp's AST is available and a code generator targets a static single assignment (SSA)-based intermediate representation (IR) like LLVM [Adv+03], GIMPLE [Mer03] or Thorin (see Part II). Translating an imperative program to SSA-form is a well-known procedure [see Cyt+91; Bra+13]. For this reason, this chapter focuses on generating code for vectorized constructs.

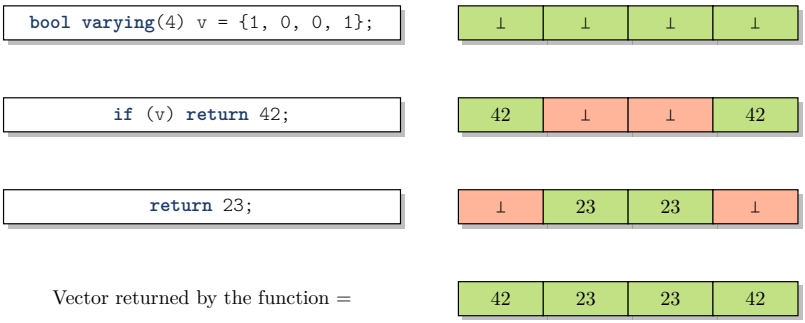
5.1. SSA Construction for Statements

To make VecImp more useful and closer to Sierra, we enhance VecImp with a *break-* and *continue-statement*:

$$
\begin{aligned}
s ::= {} & \cdots && \text{(statement)} \\
\mid {} & \texttt{break;} && \text{(break)} \\
\mid {} & \texttt{continue;} && \text{(continue)}
\end{aligned}
$$

Additionally, we allow the consequence/alternative of an *if-statement* as well as the body of a *while-statement* to end with a *return-*, *break-* or *continue-statement*.

Example 5.1 (Vectorized *Return-Statement*)
Listing 5.1 demonstrates the use of a *return-statement* in the consequence of an *if-statement*. Since the first and fourth SIMD lanes become *inactive* after evaluating the first *return-statement*, the rest of the function stays in SIMD mode and must be masked accordingly. Hence, the final *return-statement* must be combined with the former return value and the function returns `{42, 23, 23, 42}`.



Listing 5.1.: `Vectorized return`

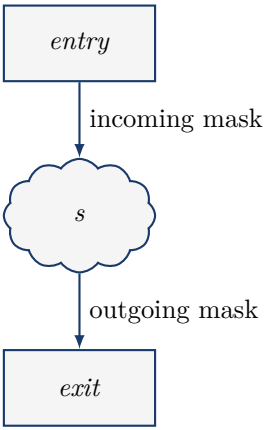


Figure 5.1.: Code generation for a vectorial statement s

For this reason, each statement conceptually produces a new *outgoing mask* because some substatement might have issued a **return**. The subsequent statement receives this mask as *incoming mask* and uses it as *current mask* in order to deactivate all SIMD lanes that have already **return**ed from the function (see Figure 5.1[1]). In addition, *break*- and *continue-statements* temporarily deactivate SIMD lanes (see Section 5.1.4). As usual, each instruction that produces a side-effect must be masked such that only active lanes are effected (see ES-Assign in Figure 4.7).

Scalar statements do not need any incoming or outgoing mask. Either a statement is executed or not. This means that whenever the program switches from scalar to SIMD mode, the code generator must create an appropriate incoming mask.

Example 5.2 (Entering SIMD mode)
Suppose the following code starts off in scalar mode:

```
bool varying(4) cond = /*...*/;
// ...
if (cond) {
    // SIMD mode
}
```

As `cond` is vectorial, so is the body of the *if-statement*. At that point, the code generator has to create an incoming mask, which is simply `cond`.

5.1.1. Nested Statements

In VecImp the only statements that have substatements are *if*- and *while-statements*. The consequence of an *if-statement* and the body of a *while-statement*, respectively, use as *current mask* the *incoming mask* combined via bitwise AND with the controlling condition. The alternative of an *if-statement* has to additionally negate the *controlling condition*.

Example 5.3 (Nested Statements)
Suppose the following code starts off in scalar mode:

[1] A cloud in this and subsequent figures represents an arbitrary control-flow subgraph whereas a box represents a single basic block.

```
int varying(4) v = {0, 1, 2, 3};
while (v >= 2) { // current mask = {0, 0, 1, 1}
    if (v % 2 == 1) {
        /*...*/  // current mask = {0, 0, 0, 1}
    } else {
        /*...*/  // current mask = {0, 0, 1, 0}
    }
}
```

As the condition in the `while` loop is of type `bool varying(4)`, VecImp enters SIMD mode. The incoming mask for the loop's body is `{0, 0, 1, 1}`. Since `v % 2 == 1` evaluates to `{0, 1, 0, 1}`, the *current mask* in the consequence of the *if-statetment* is `{0, 0, 0, 1}`. Accordingly, the *current mask* in the alternative is `{0, 0, 1, 0}`.

5.1.2. Branching on Vectorial Conditions

In order to implement vectorial versions of control-flow constructs, we often have to check whether *all* or *any* elements in a `bool varying(L)` are `true`:

- `all(v)`: Evaluates to `true` if *all* elements in `v` are `true`.
- `any(v)`: Evaluates to `true` if *any* element in `v` is `true`.

Many ISAs provide specialized instructions that implement these tests in one instruction. Armed with these insights, we now take a closer look on how to implement vectorial *if-* and *while-statements*.

5.1.3. Vectorial If-Statement

For a vectorial *if-statement* the code generator has to emit the pattern in Figure 5.2. Let us ignore the dashed edges for a moment and only follow the solid ones. This results in a linearized control flow where the *if-statement*'s entry block $entry_i$ jumps to the control flow related to the consequence. Its exit block $exit_t$ in turn jumps to the control flow related to the alternative. Its exit block $exit_e$ finally jumps to the *if-statement*'s $\texttt{exit}_i$. The $entry_i$ block sets up the incoming masks $\texttt{incoming\_mask}_t$ for the consequence and $\texttt{incoming\_mask}_e$ for the alternative by using the *if-statement*'s incoming mask $\texttt{incoming\_mask}_i$ as already discussed.

As the consequence might use **break**, **continue** or **return**, the consequence's outgoing mask $\texttt{outgoing\_mask}_t$ is not necessarily the same as its incoming mask. The same holds true for the alternative. This is why $exit_i$ has to combine a new outgoing mask $\texttt{outgoing\_mask}_i$. If neither the consequence nor the alternative uses **break**, **continue** or **return**, $\texttt{outgoing\_mask}_i$ is equal to $\texttt{incoming\_mask}_i$.

If it is likely that none of the elements in $\texttt{incoming\_mask}_t$ is **true**, the code generator can insert a dynamic check and jump as a shortcut directly to the alternative. Similarly, the code generator can check if all elements in $\texttt{incoming\_mask}_e$ are **false** and skip the alternative in that case. These shortcuts are depicted as dashed edges in the figure. Note that these additional branches require some ϕ-functions in order to properly select outgoing masks, depending on which branches have been taken.

In ISPC the programmer can use the **cif** statement, in order to give the code generator a hint that probably either all SIMD lanes are **true** or all are **false**. Note that this is more conservative than the shortcut conditions we have just mentioned. The reason for this is that ISPC generates specialized code paths that do not need masking at all.

5.1.4. Vectorial While-Statement

For a vectorial *while-statement*, the code generator has to emit the pattern in Figure 5.3. Again, let us ignore the dashed edge for a moment. The *while-statement*'s $\texttt{entry}_w$ selects via a ϕ-function as temporary incoming mask $\texttt{incoming\_mask}_t$ either the *while-statement*'s $\texttt{incoming\_mask}_w$ when first entering the loop or `outgoing_mask` otherwise. Then, the block computes the incoming mask for the body $\texttt{incoming\_mask}_b$. As long as **any** of its elements is **true**, the body must be executed. Each time a lane becomes inactive via a **break**, **continue** or **return**, the outgoing mask of the corresponding statement will be adjusted. However, the body's exit block $exit_b$ must reactivate all lanes that have **continue**d before jumping to the next iteration. For this reason, each loop maintains a boolean vector `cont_vec`, which indicates the lanes to reactivate. Thus, the final `outgoing_mask` to feed into $entry_w$'s ϕ-function is the body's outgoing mask $\texttt{outgoing\_mask}_b$ combined via a bitwise OR with that `cont_vec`. After all SIMD lanes have finished the loop, the *while-statement*'s $exit_w$ must compute $\texttt{outgoing\_mask}_w$. This is achieved by deactivating all lanes that **return**ed (indicated by the boolean vector `ret_vec`) from $\texttt{incoming\_mask}_w$.

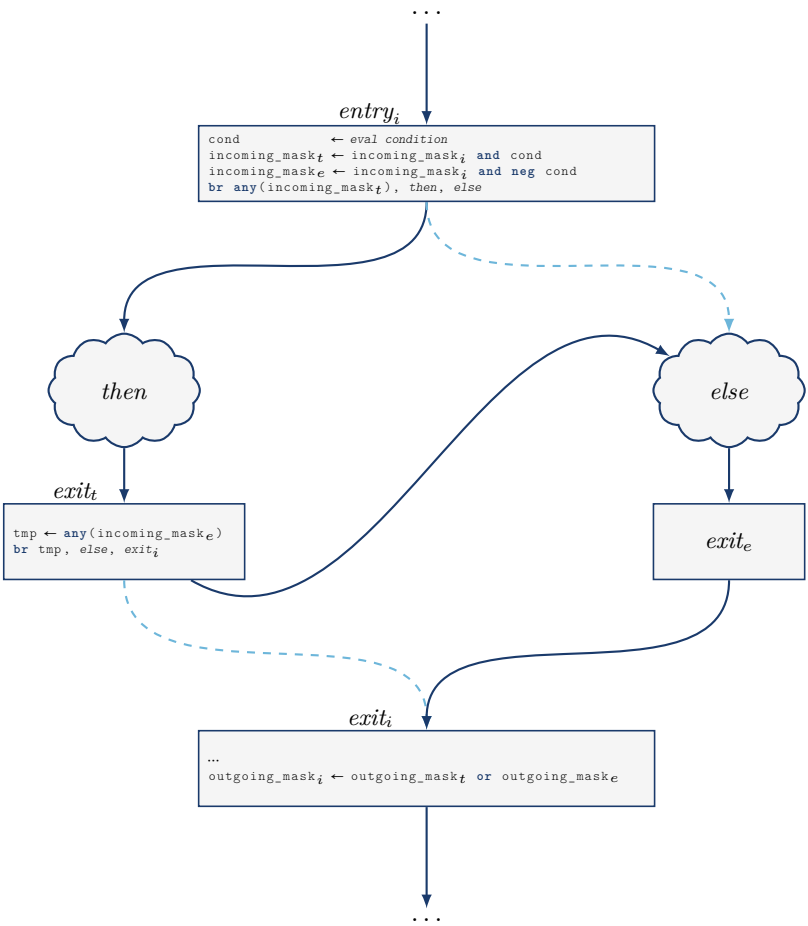


Figure 5.2.: Code generation for `if` statement

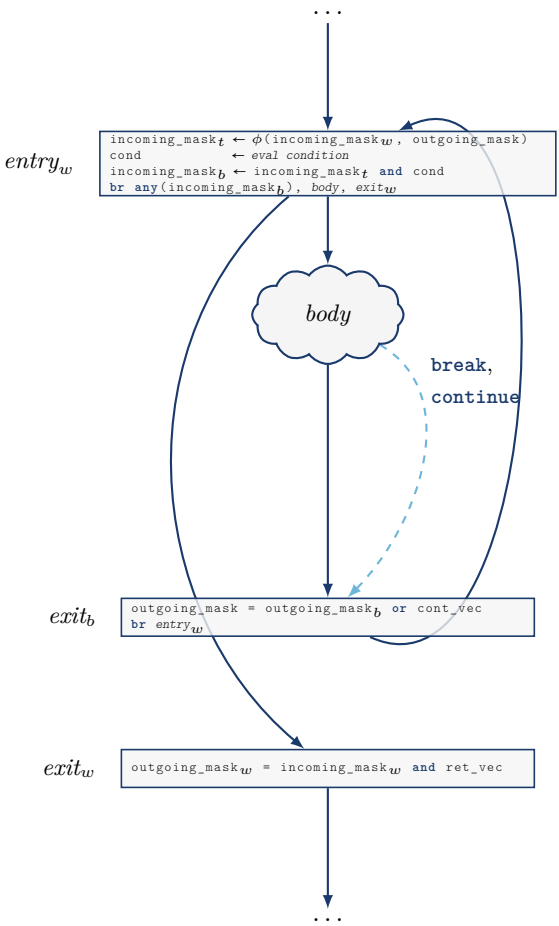


Figure 5.3.: Code generation for **while** statement

In the case that coherent control flow is likely, the code-generator may insert shortcuts to skip part of the loop body (see the dashed edge). Note that these additional branches again require some ϕ-functions in order to select the proper masks depending on which branches have been taken.

5.2. Conclusions

This chapter has discussed a syntax-directed code generator for vectorial programs. When using unstructured control flow, statements that are control-dependent on vectors are not necessarily syntactically nested inside the corresponding control-flow statement. As we have seen, supporting *break*- and *continue-statements* as well as *return-statements* in an unstructured way already makes code generation quite complicated. The use of `goto` and labels creates arbitrary CFGs. Generating valid and efficient code, requires several non-trivial analyses (see Section 2.3). These analyses are cumbersome to implement on a C++ AST. For this reason, we plan to generate scalar instead of vectorial code as future work. Then, the burden of vectorization will be shifted to a third-party vectorizer (see Section 2.3). However, this does not solve type checking since the type checker must know the current vector length of each statement. A working solution is to type-check during vectorization. This solution, however, is unsatisfactory: The quality of the error messages significantly suffers from this approach because we do not have the original AST at hand anymore. Supporting arbitrary, unstructured control flow while maintaining sophisticated error messages with a reasonable code base is an unsolved engineering problem.

"[...] Take thy beak from out my heart, and take thy form from off my door!"
— Quoth the Raven "Nevermore."
Edgar Allan Poe, The Raven

6 Evaluation

To evaluate our programming model, we first experimented with IVL in collaboration with Ingo Wald. As a C-like language, IVL is conceptually similar to VecImp and supports a simplified model of polymorphism. Furthermore, we modified an industry-strength C++ compiler to incorporate a prototypical version of Sierra.

6.1. The IVL Vectorizing Language

IVL is a prototypical compiler written by Ingo Wald [LHW12]. IVL embraces many of VecImp's concepts to generate vectorized code for Intel® Many Integrated Core Architecture (MIC) and the SSE instruction set.

6.1.1. IVL's Back Ends

MIC is a many-core x86 architecture in which each x86 core is augmented with a 16-wide vector unit accessed through a rich vector instruction set. MIC supports both scatter/gather as well as efficient masking/predication via a separate set of 16-bit mask registers. In a vectorized context, IVL emits MIC code that runs on all lanes in parallel. All *scalar* constructs are executed in the core's scalar pipe and scalar data is held in scalar registers. Rather than allowing arbitrary vector lengths, IVL only supports one vector length per translation unit. In particular, on MIC we currently support 16-wide and ("double-pumped") 32-wide vectorization—with 16 being the default. The SSE back end supports 4-wide vectorization. A successor version of IVL also supports the Intel® AVX instruction set for 8-wide vectorization.

Note that MIC and SSE instruction sets fundamentally differ in many aspects: SSE and MIC have different native vector lengths, SSE does not

support hardware accelerated scatter/gather and uses blending in order to emulate predicated execution. Given our experience in implementing the SSE and MIC back ends, we think implementing additional back ends supporting other SIMD CPU types and ISAs is straightforward.

IVL acts as source-to-source compiler: It translates IVL code to "C++ with intrinsics" code that is then passed to the Intel® C/C++ compiler (ICC) for MIC. This does not only leave all the heavy-lifting in code generation and optimization to this compiler, it also allows the programmer to visually inspect (and possibly modify) the emitted code and to use this code with tools like debuggers, performance analyzers, etc. Furthermore this means that cross-linking with other C modules is fully supported.

6.1.2. Supported Types and Language Constructs

IVL currently supports a significant subset of C and a small set of additional keywords to guide vectorization. In terms of types, IVL supports `bool`s as well as (32-bit) `int`s, `uint`s, and `float`s but only partially supports 8-bit, 16-bit, and 64-bit data types. Furthermore, IVL supports `struct`s, arrays, and references (including vector references to vectorized types) but only partially supports pointers.

As IVL currently only supports one global vector length `n` per translation unit, VecImp's `varying(n)` type qualifier is called `varying`, and `varying(1)` is called `uniform`. In terms of control flow, IVL supports all of `if`/`else`, `do`, `while`, `for`, `break`, `continue`, and `return`. In addition, IVL also supports some simple reduction operations like `all` or `any` (see Section 5.1.2).

6.1.3. Vectorization

Vectorization in IVL is done *on demand*: Similar to templates in C++, IVL parses `struct` and function definitions but does not emit code until *instances* of those types and functions are required. Vectorization of code is triggered when IVL encounters a function with the `kernel` keyword. IVL will then emit a C function for this kernel (plus some additional helper functions to allow calling this function from the host machine if required), and vectorize this kernel's body, which in turn will emit all functions called by this body on-demand, etc.

Statements and expressions operating on **varying** types will emit vector intrinsics, while purely **uniform** expressions/statements will emit only scalar code even when inside a vectorized function (like proposed by VecImp). This is highly desirable because the code will use precious vector registers and costly vector instructions only where required, and enables a mix of scalar and vector expressions that MIC's superscalar architecture (with parallel scalar U- and vectorial V-pipes) is particularly good at. For example, if a **for**-loop in a (vectorized) function uses a loop condition that only depends on a **uniform** function parameter, that respective loop control code will only use scalar x86 instructions.

6.1.4. Polymorphism

IVL supports a limited form of polymorphism. IVL instantiates poly-typed function parameters when IVL encounters the function's call site. If any argument including the hidden current mask argument is vectorial, all parameters and local poly-typed variables will become vectorial. Like templates in C++ this mechanism requires the function to be known at compile-time. To use a function across different translation units, the programmer must either explicitly specify all parameters' vector lengths or explicitly instantiate this function in one translation unit.

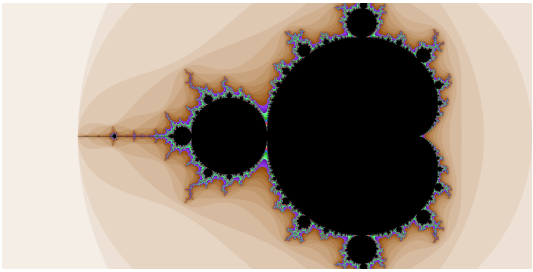
Example 6.1 (Min in IVL)
Reconsider Example 4.4. When IVL instantiates the call site `min(u,v)` where `u` is **uniform** and `v` is **varying**, IVL will use **varying** types for the parameters `a` and `b` as well as for the local variable `result` because one of the arguemts—namely `v`—is varying.

6.1.5. IVL Examples

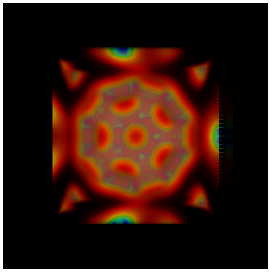
We give an overview over some examples realized with the IVL compiler, all running on a 32-core 900 MHz Knights Ferry prototype board.

Proof-of-Concept Examples

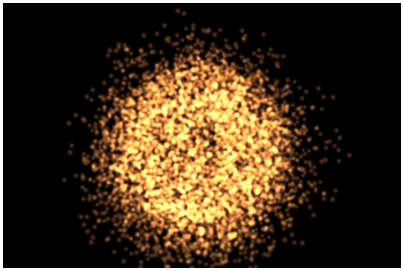
We ported `mandelbrot`, `volumerender`, and `nbody` from the CUDA SDK (see Figure 6.1a–c). These examples ran more or less "out of the box".



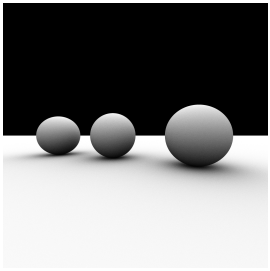
(a) `mandelbrot`



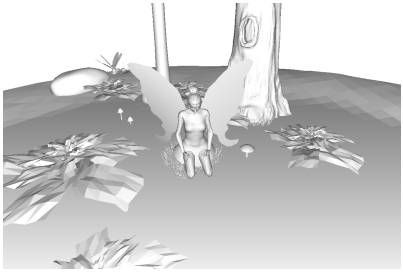
(b) `volumerender`



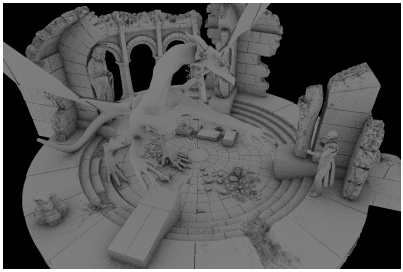
(c) `nbody`



(d) `aobench`



(e) `eyelight`



(f) `ambient occlusion`

Figure 6.1.: Several examples compiled with our prototypical IVL compiler running on an Intel® "Knights Ferry" prototype board

Since IVL does not currently support any native hardware texturing, the `volumerender` example has to resort to "manual" tri-linear interpolation to sample the volume but nevertheless already reaches roughly 30 frames per second. For all other examples, both ease of porting and resulting performance matched or exceeded expectations. As just one example, the publicly available `aobench` benchmark[1] required only trivial modification to port to IVL, while rendering a 1024×1024 frame (with 16 samples per pixel and 16 rays per sample) in 402 ms (see Figure 6.1d).

Ray Tracer Examples

As a more challenging example, we also implemented an IVL ray tracer with various shaders *into* an existing Knight's Ferry ray tracing system [Wal12]. The IVL-based traversal, intersection, and shading code was linked together with manual intrinsics code for data structure construction and other renderers. In this setting, IVL and manual C code were actually sharing the same data structures. First, we integrated an intentionally simple `eyelight` shader into the framework (see Figure 6.1e). As a next step, we included an `ambient occlusion` renderer requiring random number generation, quasi-Monte-Carlo-sampling, cumulative distribution function inversions and recursion, involving both incoherent data access patterns and SIMD lane utilization (see Figure 6.1f).

Comparison to hand-written Code

To better quantify the performance of IVL's code, we also ran some experiments where we compared IVL-generated code to manually-written reference intrinsics code. In all cases, the reference code was written *before* the IVL code. For the `eyelight` ray tracer, IVL renders a 1600×1024 frame in 13.8 million cycles, vs. 13.28 million cycles in the reference intrinsics code: a difference of only 4%. An exact reference version for the `ambient occlusion` renderer is not available, due to that code's complexity.

Finally, we also ran an artificial *k-nearest neighbor* benchmark for which we had reference code for a variety of architectures. This benchmark is highly non-trivial in both control-flow and data access patterns. In this workload, IVL requires 755 million cycles (for 1 million 50-neighbor queries in a 1 million point dataset), as compared to 656 million cycles for the

[1] see `https://code.google.com/archive/p/aobench/`

hand-coded version—a difference of only 15%. The IVL version is also roughly two orders of magnitude faster than (single-threaded) scalar host code, and roughly twice as fast as a CUDA version of that kernel running on a Fermi GT480 GPU.

The performance gap is mostly encumbered by the fact that IVL is a source-to-source compiler emitting vectorized code and relies on ICC for optimizations. This compiler does not yet recognize some patterns that are suitable for optimizations but would have been applied by a human intrinsics programmer.

6.2. Sierra

We also implemented a prototype of our proposed C++-extension.[2] The Sierra compiler is a fork of the LLVM-based compiler Clang 3.3, and thus, supports the complete C++11 standard. The extension must be explicitly enabled via the switch `-fsierra`.

6.2.1. Supported Types and Language Constructs

In contrast to IVL or ISPC, Sierra supports fine-grained control over the vectorization length as in VecImp. All integer and floating-point types can be qualified with `varying(n)`. If no `varying(n)` qualifier is given, Sierra defaults to usual scalar semantics. Thus, activating the Sierra extension will not break any existing C++ code: Without using any Sierra types, the Sierra compiler is still a usual C++ compiler. Furthermore, Sierra has limited support for `varying struct` variables. Like in ISPC or IVL, vectorization is recursively applied to all fields.

In terms of control flow, the Sierra compiler supports `if`/`else`, `while`, `for`, `break`, `continue`, `return`, and short circuit evaluation. Although Sierra does not directly support polymorphism, it can be mimicked to a certain degree with C++ templates (see Example 3.8).

Example 6.2 (Min in Sierra)
Reconsider Examples 4.4 and 6.1. Sierra can mimic IVL's behavior by using one template parameter:

[2] see `http://sierra-lang.github.io/`

```
template<int L>
simd(L)
int varying(L) min(int varying(L) a, int varying(L) b) {
    int varying(L) result = a;
    if (b < a)
        result = b;
    return result;
}
```

To some extent Sierra can also mimic PolyVecImp's behavior by using one template parameter for each argument. However, as template parameter deduction in C++ only happens locally, the programmer must manually encode the math to derive a suitable vector length:

```
template<int L, int A, int B>
simd(L)
int varying(max(A,B)) min(int varying(A) a, int varying(B) b) {
    int varying(max(A,B)) result = a;
    if (b < a)
        result = b;
    return result;
}
```

6.2.2. Implementation

Sierra compiles arithmetic vector types to LLVM vector types. During its type legalization phase [BR13] LLVM in turn splits vectors to the machine's native vector length if necessary. In particular, this allows for double-pumping, e.g., using `float varying(8)` on SSE.

As outlined in Chapter 5, Sierra vectorizes code from its AST representation. Consequently, Sierra directly emits vectorized LLVM code. From there on, Sierra runs Clang's default driver to steer the LLVM pipeline. Sierra does not rely on any specific LLVM patches.

6.2.3. Sierra Examples

We implemented several programs in order to evaluate the performance of Sierra. As demonstrated in Section 1.2, we reuse the same program for all variants: We exposed the desired vector length as macro, such that passing `-DVECTOR_LENGTH=L` via command line sets the vector length of

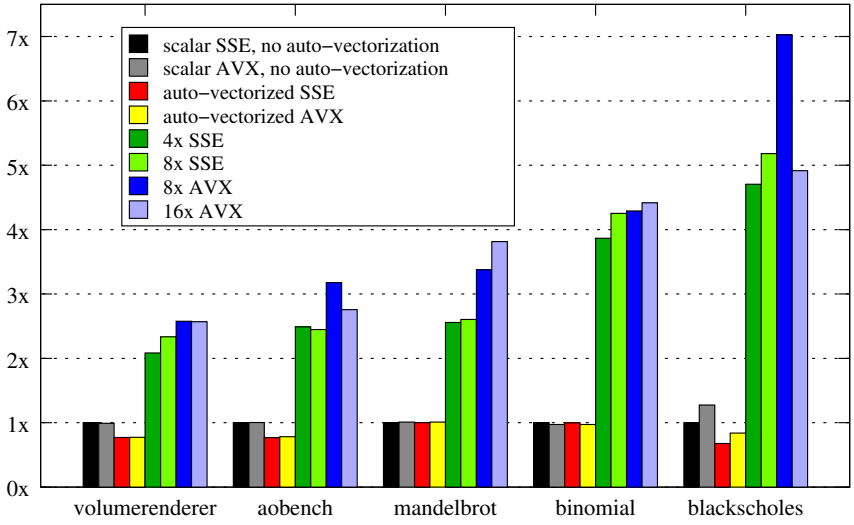


Figure 6.2.: Speedups compared to the scalar SSE version

the benchmark to `L`. No changes to the code were necessary to create the variants. We compiled all programs with `-O3` and `-ffast-math` to allow for further optimizations. We tested our programs with SSE 4.2 (`-msse4.2`) and AVX (`-mavx`).

Our test ran on an Intel® Ivy Bridge Core™ i7-3770K CPU. We used the median performance of 11 runs for computing the speedups shown in Figure 6.2.

Like in the IVL evaluation, we ported the publicly available `aobench`. Furthermore, we also ported `volumerender` (see also Section 1.2), `mandelbrot`, the `binomial` options pricing model, and the `blackscholes` algorithm from the CUDA SDK.

First, we measured the performance of scalar programs *without* using LLVM's built-in auto vectorization.[3] This scalar variant compiled for SSE serves as baseline for all other variants of the same program. Consequently, all non-auto-vectorized SSE programs have a speedup of 1x. Next, we explicitly enabled LLVM's auto-vectorizer for all programs. Then, we

[3]This can be controlled via `-fno-vectorize`, `-fno-slp-vectorize` and `-fno-slp-vectorize-aggressive`.

instantiated vectorized versions. In the case of SSE, we instantiated variants with vector length 4 (native) and 8 (double-pumped). In the case of AVX, we instantiated variants with vector length 8 (native) and 16 (double-pumped).

Without using any vectorization techniques, compiling for AVX instead of SSE did not make any notable difference except for `blackscholes` that ran slightly faster. Surprisingly, auto-vectorization either did not affect the runtime at all or even imposed a performance penalty. The loop vectorizer was never triggered; the SLP vectorizer introduces an overhead for pooling scalars into vectors; finally, the CPU's SIMD unit might cause slowdowns if it is only barely used. Using Sierra's 4x vectorization on SSE resulted in a speedup of roughly 2x for `volumerender`, 2.5x for `aobench` and `mandelbrot`, almost 4x for `binomial`, and about 4.5x for `blackscholes`. Double-pumping yielded a small improvement most of the time. Using Sierra's 8x vectorization on AVX resulted in a speedup of roughly 2.5x for `volumerender`, 3x for `aobench`, 3.5x for `mandelbrot`, 4x for `binomial`, and 7x for `blackscholes`. We obtained mixed results when double-pumping AVX. We believe this is due to the fact that AVX is internally already double-pumped on Ivy Bridge. Moreover, many AVX instructions still use a native vector length of 4 instead of 8.

6.2.4. Further Improvements

Overall, LLVM's generated code is of modest quality. A major dilemma is that most ISAs are unclear about the exact representation of boolean vectors. For example, comparing two `float varying(4)` values actually yields a `uint32_t varying(4)` on SSE. Each `uint32_t` component represents a mask consisting of either `0` or `~0`. Special blend instructions (or bit arithmetic on older SSE versions) use these masks as input to implement the masking for vectorized control flow (see Section 3.2). But a comparison of `double varying(4)` values yields a `uint64_t varying(4)`. The reason for this is that this comparison is double-pumped on SSE. In Sierra, all comparisons yield boolean vectors, which get translated to boolean vectors in LLVM. But LLVM's representation for boolean vectors is a consecutive sequence of bytes. Indeed, LLVM tries to eliminate conversions, but currently this only works on a per-basic-block level. Thereby, LLVM introduces superfluous conversions, which additionally increase register pressure.

A related problem is the implementation of `any` and `all` (see Section 5.1.2). SSE and AVX offer special instructions for this, but it is difficult to provoke

the emission of these instructions in LLVM. The same is true for other patterns that are mappable to built-in assembly instructions of the ISA. For instance, AVX supports an instruction to find the minimum of two `float` vectors. The front end could directly emit the machine instruction in question (via an LLVM intrinsic), but this is a mixed blessing. On the one hand, one can be sure that the intended instruction is selected during code generation. On the other hand, LLVM' analyses and transformations do not know the semantics of these intrinsics. Even simple transformations like constant folding usually don't work on intrinsics.

Furthermore, LLVM's analyses and transformations are just not as sophisticated for vectors as for scalars in many areas. Additionally, special transformations may be needed in order to use some tricks an experienced human intrinsic programmer would have used.

For these reasons, we believe that there is still much room to improve the performance of Sierra's emitted vectorized code. Moreover, we hope that AVX-512, which is similar to MIC's SIMD instruction set, will solve many of these problems. AVX-512 introduces a special predication register file that resolves the discussed ambiguity for boolean vectors.

The Ultimate Answer to Life, The Universe and Everything is... 42!

Douglas Adams,
The Hitchhiker's Guide to the Galaxy

7 Conclusions

This part of the thesis presents a SIMD extension for C++. Although this extension focuses more on the C subset of C++, it integrates well with many C++ features like templates. Our implementation demonstrates that such an extension is effective while our benchmarks back that SIMD is too important to be ignored.

Furthermore, Sierra is in the spirit of C++ [Str07]:

- Explicit vector types provide predictable performance gains on SIMD hardware.
- Vector types are portable.
- Vectorization of data types provides the programmer with a tool to build SIMD-friendly data structures.
- Automatic masking massively eases programming and makes vector code almost look like scalar code. For this reason, Sierra is attractive for existing C++ projects that want to adopt SIMD.

On the downside, actually implementing Sierra is an intrusive venture because Sierra interacts with many aspects of the C++ language and C++ is already very complex in itself. Furthermore, albeit Sierra already takes much work out of the programmer's hands, she still has to manually annotate many types with `varying`. Ideally, the programmer would only instruct the compiler to enter SIMD mode at some point in the code, and the compiler would automatically transform the code as necessary—including the inference of `varying` annotations. Finally, non-SIMD accelerators are not supported by Sierra. In order to address these, the programmer must resort to other techniques like OpenCL, OpenACC, or OpenMP. Part II of this thesis addresses all these concerns.

Part II.

AnyDSL
Building Domain-Specific Languages for Productivity and Performance

When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.

Edsger W. Dijkstra

8 Introduction

To achieve optimum performance, programs have to be transformed in a way that is beyond the scope of ordinary compiler optimizations. These transformations have two goals: First, to exploit domain knowledge, which is lost in the implementation and not accessible to the compiler. Second, to utilize features of the target hardware architecture to improve performance (vectorization, memory hierarchy, etc.).

One way to achieve this performance is to design a DSL that provides language constructs to express domain knowledge. A special DSL compiler can use this knowledge to generate highly-optimized code for a specific architecture. A popular approach to implement a DSL is to *embed* the DSL into a *host* language. This allows the DSL designer to reuse the lexer, parser and type checker of the host language.

8.1. Deep vs. Shallow Embedding of DSLs

Gibbons and Wu [GW14], for instance, distinguish between *deep* and *shallow embeddings*.[1] We discuss the differences of these styles by means of the domain-specific construct `range` (see Listing 8.1). This construct iterates from `a` (inclusive) to `b` (exclusive) and executes `body` each time whereby `i` serves as induction variable.

8.1.1. Deep Embedding

In a deep embedding, the application developer writes a program generator `p_gen`. This generator is the embedded DSL program (see Listing 8.2a). Executing this generator creates a representation `p_spec` of the program

[1] Boulton et al. [Bou+92] were likely the first who coined these terms.

```
for i in range(a, b) {
    body
}
```

Listing 8.1.: Domain-specific `range` loop

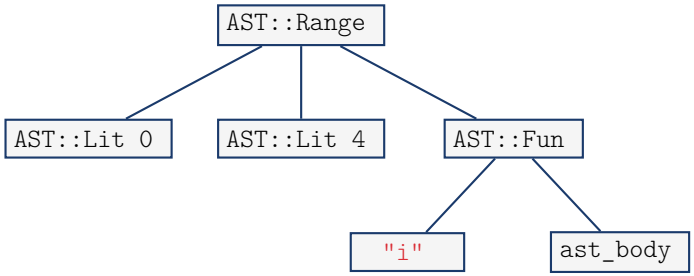


Figure 8.1.: Program representation `p_spec` of Listing 8.2a

as data structure (see Figure 8.1) available in the host language H. Furthermore, the generator can be made parametric in certain aspects of the embedded DSL program. Thus, executing this program *partially evaluates* the embedded program with respect to the given input. For example, we can construct an unrolled program representation:

```
let p_spec = unroll(0, 4, fun("i", ast_body));
```

It is straightforward to implement an interpreter for `p_spec` (see Listing 8.2b). Note how the interpreter for the deep embedding needs to match the returned values of `eval` within the `AST::Range` case in order to make sure that the returned values are properly tagged. This issue is known as the *tag problem* [CKS07, §1.1]. However, in order to achieve performance, we need to compile the program. Therefore, we have to write a code generator that inspects `p_spec` and emits code for some language (like C or LLVM). Since `p_spec` is a data structure, deep embeddings allow for powerful, domain-specific optimizations [e.g. Cha+10; Rag+13; Klo+14; Rat+17], which can be applied beforehand. Note that `p_gen`, the optimizer, and the code generator are all written in H.

In terms of programming experience, one drawback of deep embeddings is that the application developer actually writes a program generator instead of a program. Modern deep embedding frameworks alleviate this problem

```
// DSL implementation
fn range(a: AST, b: AST, body: AST) -> AST {
    AST::Range{a, b, body}
}
//...

// DSL program
//...
let p_spec = range(lit(0), lit(4), fun("i", ast_body));
```

(a) Deeply embedded DSL program `p_gen` to construct `p_spec`

```
fn eval(ast: AST) -> Val {
  match ast {
    AST::Range{ast_a, ast_b, ast_fun} => {
      match (eval(ast_a), eval(ast_b), eval(ast_fun)) {
        (Val::Int{a}, Val::Int{b}, Val::Fun{fun}) => {
          let mut i = a;
          while i < b {
            apply(fun, i);
            ++i;
          }
          Val::Unit
        },
        _ => error("type error")
      }
    },
    AST::Lit{i} => Val::Int{i},
    // other cases
  }
}
```

(b) Interpreter for a deeply embedded DSL. The environment to track variable bindings has been elided.

Listing 8.2.: Deep embedding of `range`. For the sake of simplicity, some pointer indirections, which would be needed in actual code, have been elided.

```
// DSL implementation
fn range(a: int, b: int, body: fn(int) -> ()) -> () {
    let mut i = a;
    while i < b {
        body(i);
        ++i;
    }
}

// DSL program
//...
range(0, 4, |i| body);
```

Listing 8.3.: Tagless interpreter for a shallowly embedded `range`. The expression `|i| body` means λ`i.body`.

by *virtualizing* the host language [Cha+10]: Via clever overloading of *H*'s language constructs, an *H*-expression like `a + b` does *not* perform an addition; instead, executing this expression constructs a domain-specific program representation that *represents* this addition. This virtualization is intrinsically not entirely faithful and compromises the illusion of actually writing a DSL program in several ways [Jov+14]:

- The host language cannot be virtualized entirely. For example, while many languages allow overloading the +-operator, only few provide a mechanism to overload a `while`-loop. What is more, overloading a `continue`-statement is not possible per se because of its unstructured nature. See Section 8.2.1 below for our take on this problem.

- On the other hand, implementation details of the embedded DSL leak to the application developer. For example, the integer type of the DSL is the type of an appropriate AST node instead of just `int`.

- This immediately leads to the effect that error messages are more difficult to understand for the application developer. Even worse, the embedded program may be a well-typed *H* program but an ill-typed DSL program. In this case, the error message will not be emitted during the compilation of the *H* program; instead, the error message will appear when running `p_gen`.

- While debugging tools for H are usually available, the tool support for the constructed program `p_spec` is usually rather modest. Each new domain-specific program representation on which `p_gen`, the optimizer, and the code generator work, must have support for debugging. Even if those tools were available, the application developer still would still have to figure out which parts of his program run at which stage and how to properly instrument these tools at the given stage.

- Although the compiler for H already features a code generator, the DSL designer has to write a different code generator that emits code for the domain-specific program representation `p_spec`.

- To reason about the embedded program `p_spec`, the application developer ultimately has to understand how the generator `p_gen` works.

8.1.2. Shallow Embedding

In a shallow embedding the domain-specific constructs are defined by simply implementing their semantics in the host language (see Listing 8.3). This version semantically resembles the interpreter in Listing 8.2b but does not suffer from the tag problem. For this reason, Carette, Kiselyov, and Shan [CKS07] call this approach a *tagless interpreter*. The application developer *directly* writes the embedded program in language H.

In contrast to deep embedding, shallow embedding cannot manipulate the embedded program since it is not available as a data structure. For this reason, shallowly embedding a high-performance DSL (e.g. HIPA^{cc} [Mem+16] or SYCL [Khr15]) involves the unpleasant task of modifying an existing compiler.

8.2. The AnyDSL Way

In this part of the thesis we present the *AnyDSL* framework. AnyDSL consists of the language Impala and its IR Thorin (see Section 8.2.2 below). Impala offers both imperative as well as functional programming and thus is an attractive host language for *shallow* DSL embedding. Embedding a DSL into Impala means that the DSL designer implements a domain-specific

```
fn main() {
    let g = stencil(jacobi, field);
}
```

(a) Application developer

```
fn stencil(s: Stencil, field: Field) -> Field {
    let mut out: Field = { /* ... */ };

    for x, y in @iterate(out) {
        out.data(x, y) = apply_stencil(x, y, field, s);
    }
    out
}
```

(b) DSL designer

```
fn iterate(field: Field, body: fn(int, int) -> ()) -> () {
    let grid = (field.cols, field.rows, 1);
    let block = (128, 1, 1);
    with nvvm(grid, block) {
        let x = nvvm_tid_x() + nvvm_ntid_x() * nvvm_ctaid_x();
        let y = nvvm_tid_y() + nvvm_ntid_y() * nvvm_ctaid_y();
        body(x, y);
    }
}
```

(c) Machine expert

Listing 8.4.: Dissecting performance-critical code into layers of abstractions using AnyDSL

construct as a tagless interpreter—just like in Listing 8.3. As calling higher-order functions like this is very common, Impala features a `for`-expression as syntactic sugar such that the application developer can instead write:

```
for i in range(0, 4) { body }
```

In order to eliminate any overhead that shallow embedding imposes, Impala features a partial evaluator, which the programmer can steer with a few annotations.

Let us now see how everything fits together in the case of an image processing DSL (see Listing 8.4). The application developer (see Listing 8.4a) uses domain-specific constructs such as applying a Jacobi stencil to a `field`. He does not need to know the implementation details of these constructs. The DSL designer (see Listing 8.4b) provides implementations for these constructs, which in turn use certain abstractions: a function that `iterates` over the `output` field and a convolution operation on the field point. The convolution is parameterized by the concrete coefficients of the stencil `s`. A machine expert then provides an optimized implementation of these primitives for each target platform. See Listing 8.4c for an implementation of `iterate` tuned for execution on an NVIDIA GPU. Similarly, a different machine expert might provide an implementation for `iterate` which triggers SIMD vectorization by using the built-in function `vectorize` (see Listing 10.11 and Section 10.7.1). Note how `stencil` resembles a tagless interpreter in the sense that it "interprets" `s` on the `field`. In order to completely remove the overhead of the interpreter, the DSL designer invokes the partial evaluator (controlled by the `@`-operator) to link the individual components together. The resulting code looks just as if a programmer had written it precisely for the Jacobi stencil and optimized it for an NVIDIA GPU. Section 10.7.2 presents the sketched DSL for stencil computations in more detail and discusses its performance.

This paradigm addresses the concerns of Chapter 7:

- The shallow embedding does not require to implement or modify a compiler.
- The function that is passed to the built-in `vectorize` function enters SIMD mode and the programmer does not need to annotate types with `varying`.
- With the same mechanism, AnyDSL supports different accelerators (see Section 10.7.1).

Using partial evaluation, we are limited to optimizations that can be expressed by *specializing* code. Optimizations that analyze and rewrite programs are not possible without modifying Impala's compiler. We argue that many of such optimizations can be expressed by proper abstractions in the style of `iterate`. For example, one common objection is: "How do you fuse loops?". The V-Cycle DSL in Section 10.7.3 demonstrates how to pass loop bodies as functions and invoke them in a single fused loop. Other optimizations that cannot be expressed in this way have to be implemented on Thorin, which is sufficiently high-level to facilitate this. For the DSLs we present in Section 10.7, we did not have to modify Impala's compiler.

8.2.1. Continuations

To allow for non-trivial control flow in embedded DSLs, Impala embraces *continuation-passing style (CPS)*. As Appel [App06, p.2] aptly puts it:

> Continuation-passing style is a program notation that makes every aspect of control flow and data flow explicit. It also has the advantage that it's closely related to Church's λ-calculus, which has a well-defined and well-understood meaning.

Continuations are functions that never return; think of a parameterized code block that can be passed around as a first-class citizen and must end again with a call to another continuation. CPS allows us to *uniformly* represent control flow by continuations: jumps to basic blocks, function calls, generators, exceptions, etc.

Impala represents all control flow (including functions) as continuations. In fact, *direct-style* functions and function calls—the usual style of programming—are only syntactic sugar for their CPS counterparts. Listing 8.5 showcases Impala's internal CPS representation of a `for`-expression that uses unstructured control flow. Note that `break`, `continue`, and `return` are not actually keywords. Impala will usually name the return parameter of a function "`return`" if the programmer uses direct style. However, when using a `for`-expression, Impala names the return parameter of the passed continuation "`continue`". On the one hand, this is more appropriate for the semantic effect when calling this continuation. On the other hand, `continue` will not shadow the implicitly declared `return` parameter of the contained function. Similarly, `break` denotes the return continuation that is passed to `range` when using the `for`-expression. This

makes `break`, `continue`, and `return` first-class citizens, which can be captured, invoked, or passed to other functions. This allows the DSL designer to build her own sophisticated *generators* while still supporting (potentially multi-leveled) `break` and `continue`—or even pass them around—as Listing 8.6 demonstrates. This mechanism makes Impala's `for`-expression already quite powerful. But since the DSL designer can use continuations as she likes, she can invent completely original control-flow patterns. See Section 10.7.2 for an example.

8.2.2. Thorin

Higher-order programs are deemed to be slow because the standard technique to implement a function value with free variables is a *closure*: a data structure that must be allocated at runtime and consists of a pointer to a function and bindings of all free variables. Ideally, Listing 8.5 should be compiled into a simple loop without the need to allocate a closure. Even Listing 8.6 can be compiled into two nested loops if g is inlined and does not pass `continue` to yet another function.

Impala compiles the input program into Thorin (see Listing 8.5b). Thorin only knows three concepts: continuations, their parameters, and direct-style primitive operations (primops) like +, −, etc. This makes Thorin straightforward to use and analyse. Yet, Thorin allows to directly represent higher-order functions and is thus more powerful and expressive than classic first-order compiler IRs like LLVM [Adv+03] or GIMPLE [Mer03].

Thorin employs an aggressive closure elimination phase, which optimizes typical uses of higher-order functions—like generators—in a way such hat no expensive closures are needed at runtime. Furthermore, the above-mentioned partial evaluator is also implemented on top of Thorin.

8.3. Contributions

This part of the thesis makes the following contributions:

- Chapter 9 discusses Thorin formally. We show how to translate Thorin to λ^{cps}, a CPS-based variant of PCF [Plo77], because it is more elegant to formally argue about λ^{cps} than directly about Thorin. On top of λ^{cps}, we introduce a type system as well as an indeterministic reduction system and prove type preservation and confluence.

```
fn f(a: int, b: int) -> () {

  for i in range(a, b) {
    if i == 23 {
      continue()
    } else if i == 42 {
      break()
    } else if i == 97 {
      return()
    }
    print(i)
  }
}
```

(a) Original direct-style Impala code

```
f(a: int, b: int, return: cn()):
    range(a, b, body, break)
body(i: int, continue: cn()):
    br(i == 23, A, B)
A(): continue()
B(): br(i == 42, C, D)
C(): break()
D(): br(i == 97, E, F)
E(): return()

F(): print(i, K)
K(): continue()
break(): return()
```

(b) Internal CPS representation (Thorin)

```
fn f(a: int, b: int, return: fn() -> !) -> ! {
    fn body(i: int, continue: fn() -> !) -> ! {
        if i == 23 {
            continue()
        } else if i == 42 {
            break()
        } else if i == 97 {
            return()
        }
        fn K() -> ! { continue() }
        print(i, K)
    }
    fn break() -> ! { return() }
    range(a, b, body, break)
}
```

(c) Desugared Impala program in CPS. Note that `-> !` acknowledges the fact that continuations do not return.

Listing 8.5.: Impala blurs the line between direct style and CPS. `cn(T)` denotes the type of a continuation expecting an argument of type `T`. Direct-style functions do not exist in the CPS presentation.

```
// This generator allows the user to skip several iterations
// by feeding an appropriate skip amount into 'continue'.
// The generator returns how many iterations have been performed.
fn sum_range(mut a: int, b: int, body: fn(int) -> int) -> int {
    let mut sum = 0;
    while a < b {
        ++sum;
        a += body(a)
    }
    sum
}

fn f(/*...*/) -> int {
    let sum = for i in sum_range(0, 23) {
        // capture 'break' continuation of outer loop
        let outer_break = break;

        if condition_i {
            continue(3) // skip 3 iterations
        }

        // pass 'continue' continuation to some function
        g(continue);

        for j in range(a, b) {
            if emergency_condition {
                outer_break(-1) // break both loops with sum -1
            }

            if condition {
                return(42) // leave f completely with result 42
            }

        }
        1 // default: only advance one iteration
    }
    sum
}
```

Listing 8.6.: Unstructured, higher-order control flow in Impala using first-class continuations

- Chapter 10 presents a pragmatic algorithm for partial evaluation—the technique we use to remove the overhead of shallowly embedded DSLs. We present our partial evaluation algorithm as a deterministic reduction system—a subset of the aforementioned reduction relation. This allows us to prove our partial evaluator correct and formally describe a termination property of it.

  Furthermore, we show how mapping to different hardware accelerators can be neatly expressed by higher-order functions. Our approach allows to weave in platform-specific mapping strategies such as executing code on a GPU or vectorizing code for a CPU by compiler-known, higher-order functions. We demonstrate that our partial evaluation approach enables an efficient shallow embedding of high-performance DSLs for visual and high-performance computing in Impala.

- Chapter 11 discusses how to generate code for Thorin programs. We present a novel, aggressive closure elimination, which relies on a simple, yet versatile transformation: *lambda mangling*. Our experiments evaluate Thorin on *The Computer Benchmark Game*, a set of small programs to benchmark performance across various programming languages—imperative as well as functional ones. We show that the Impala programs, which use higher-order functions, match the performance of the corresponding C implementations. Finally, we employ software engineering statistics to back that code transformations for Thorin programs are actually easier to implement than for a classic SSA-based CFG representation, although Thorin programs are more expressive because Thorin supports higher-order functions.

Since the Chapters 9–11 cover quite different topics, each chapter discuses relevant related work on its own.

8.4. Publications

The work in this part is based upon the following publications:

- Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. “Simple and Efficient Construction of Static Single Assignment Form”. In: *Compiler Construction - 22nd International Conference, CC 2013, Held as Part of*

the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. 2013, pp. 102–122. DOI: 10.1007/978-3-642-37051-9_6.

- Marcel Köster, Roland Leißa, Sebastian Hack, Richard Membarth, and Philipp Slusallek. "Code Refinement of Stencil Codes". In: *Parallel Processing Letters* 24.3 (2014). DOI: 10.1142/S0129626414410035.

- Richard Membarth, Philipp Slusallek, Marcel Köster, Roland Leißa, and Sebastian Hack. "Target-specific refinement of multigrid codes". In: *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC '14, New Orleans, Louisiana, USA, November 16-21, 2014*. 2014, pp. 52–57. DOI: 10.1109/WOLFHPC.2014.5.

- Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. "Shallow embedding of DSLs via online partial evaluation". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2015, Pittsburgh, PA, USA, October 26-27, 2015*. Best Paper Award. 2015, pp. 11–20. DOI: 10.1145/2814204.2814208.

- Roland Leißa, Marcel Köster, and Sebastian Hack. "A graph-based higher-order intermediate representation". In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*. 2nd place: Artifact Evaluation for CGO/PPoPP'15. 2015, pp. 202–212. DOI: 10.1109/CGO.2015.7054200.

Some paragraphs of these publications appear verbatim in this thesis. This applies in particular to Sections 9.1, 10.2, 10.7, 11.3, and 11.4.

"Hush!" said Gandalf. "Let Thorin speak!"
And this is how Thorin began.

J.R.R. Tolkien, The Hobbit

9 Thorin and λ^{cps}

This chapter first recaps properties of existing IRs and thus motivates the design of Thorin. We also recommend Appel's [App06, §1.2] discussion of several IRs from various points of view. In contrast to existing CPS representations, Thorin is characterized by the fact that it does *not* have an explicit scope nesting. This chapter discusses exactly what that means and what benefits it has. However, for a formal discussion, an explicit scope nesting is very helpful. For this reason, we introduce λ^{cps}—a variant of Thorin *with* explicit scope nesting. Furthermore, we introduce a type-preserving and confluent reduction system for λ^{cps}. This machinery will be needed for the next chapter when discussing partial evaluation.

The notation used is described in Appendix A; more extensive proofs can be found in Appendix B.

9.1. Related Work

9.1.1. The λ-Calculus

The λ-calculus [Chu32] is a minimalistic formal system that expresses computations. We refer readers interested in the history of the λ-calculus to Cardone and Hindley [CH06]. As the λ-calculus is syntactically very simple and represents higher-order functions naturally, IRs for functional languages are often based upon the λ-calculus. For example:

- GHC's Core [TC10; CR14] is used for type inference, type checking, and several optimizations before translating Core to a lower level language.
- The OCaml compiler uses its Lambda IR [MMH13, chap. 23] for several optimizations.

- Lift [SRD17] is a data-parallel, functional IR, which encodes OpenCL-specific constructs as functional patterns.

Name Capture

A well-known problem when transforming programs is *name capture* [Bar84].

> **Example 9.1** (Name Capture)
> The following naïve β-reduction of λx is incorrect:
>
> $$\lambda a.(\lambda x.(\lambda a.x))\ a \quad \Rightarrow \quad \lambda a.\lambda a.a$$
>
> Originally, the variable a referred to the outer λa, but now it refers to the inner λa.

One solution is to introduce new names during β-reduction:

$$\lambda a.\lambda a'.a$$

This technique is used by the inliner of GHC, for example [PM02]. The downside is that additional bookkeeping in the compiler is necessary.

De Bruijn [De 72] indices (not to be confused with De Bruijn [De 80] notation) provide another option. Each occurrence of a variable is replaced by an index. This indicates the number of binders in scope between the occurrence and its associated binder.

> **Example 9.2** (De Bruijn Indices)
> Using De Bruijn indices we rewrite Example 9.1 as follows:
>
> $$\lambda.(\lambda.(\lambda.2))\ 3 \quad \Rightarrow \quad \lambda.\lambda.2$$

De Bruijn indices have the advantage that they obviate α-conversion.

> **Example 9.3** (α-Equivalence)
> Using De Bruijn indices we identify both
>
> $$\lambda a.(\lambda b.a) \quad \text{and} \quad \lambda x.(\lambda y.x) \quad \text{with} \quad \lambda.1(\lambda.2)\ .$$

A disadvantage of De Bruijn indices is that these indices depend on the context in which they occur. For instance, in the latter example both indices refer to the same binder. Analyses have to keep track of the binding level in order to figure out which indices identify. Furthermore, code transformations, for example β-reduction, must adjust indices.

Yet another alternative is to give up on names and use a graph. Each occurrence directly points to its definition:

It is now straightforward to perform a β-reduction. Thereby, it is not necessary to rename anything: We simply update the body of the inner function to point to the substituting argument. This way, the reference in the substituting argument to the outer function still points to its proper definition.

9.1.2. Continuation-Passing Style

Although the λ-calculus' simplicity makes this language an attractive IR, it also has its downsides. Most notably, we must be careful during β-reduction.

> **Example 9.4** (Dangerous β-reduction)
> Reducing
> $$(\lambda x.23)(fy) \quad \Rightarrow \quad 23$$
> is problematic if f diverges or has side effects—assuming an extension of the λ-calculus that allows this.

The reason is that the λ-calculus does not impose a strict evaluation order. CPS on the other hand establishes a well-defined evaluation order. Furthermore, CPS allows us to represent *any* control flow in a *uniform* way. Section 8.2.1 has already discussed, how to model basic blocks, ordinary jumps, loops, and generators with CPS. As another example, consider how Listing 9.1 models Java-style checked exceptions in Impala without syntax support. These properties make CPS an attractive IR and, thus, CPS has been used in several compilers [e.g. Ste78; Kra+86; App06].

```
fn fopen(name: string, throw: fn(FileError) -> !) -> File {
    if sth_went_wrong {
        throw(FileError{/*...*/})
    }
    File{/*...*/}
}

//...
let file = fopen("some_file", catch);
// ...
fn catch(file_error: FileError) {
    // error handler
}
```

(a) Java-style checked exceptions in Impala

```
fopen(name: string, throw: cn(FileError), return: cn(File)):
    br(sth_went_wrong, T, F)
T():
    throw(FileError{/*...*/})
F():
    return(File{/*...*/})

//...
    fopen("some_file", catch, k);
k(file: File):
    //...
catch(file_error: FileError):
    // error handler
```

(b) Thorin representation

Listing 9.1.: Checked exceptions modeled in CPS

```
+(a, b, AB): where
  AB(ab):
    +(c, d, CD) where
      CD(cd):
        *(ab, cd, R)
```

(a) compute a+b first

```
+(c, d, CD) where
  CD(cd):
    +(a, b, AB): where
      AB(ab):
        *(ab, cd, R)
```

(b) compute c+d first

Listing 9.2.: The order of computing ab and cd does not matter as long as the multiplication happens last.

Since invoking a continuation does not return, we have to equip it with an additional higher-order "return" parameter in order to mimic direct-style functions in CPS. At the call site, this parameter is fed an appropriate continuation that retrieves the result. This process is called *CPS conversion*.

Example 9.5 (CPS Conversion)
CPS converting Example 9.4 yields

$$f\ y\ (\lambda r_f.(\lambda x.\lambda r.r\ 23)\ r_f\ R)$$

where R represents the rest of the program. If we allow ourselves functions of multiple variables and a bit of syntactic sugar, we can rewrite the example like this:

```
f(y, k_f) where
  k_f(r_f):
    g(r_f, R) where
      g(x, r):
        r(23)
```

Note how CPS conversion turns function applications inside out and names all intermediate results of computations.

The fixed evaluation order in CPS is both a curse and a blessing. On the one hand, the compiler does not have to worry about side-effects and non-termination during β-reduction. On the other hand, a fixed evaluation order suggests false dependencies between calculations. This complicates *code motion* (see Listing 9.2), *common subexpression elimination*, and many other optimizations. Other arguments against CPS are the complexity

```
a(x: int, ret: int -> ⊥) -> ⊥:
  b() where
    b() -> ⊥:
      c() where


        c() -> ⊥:
          ret(x)
```

(a) Classic CPS version

```
a(x: int, ret: int -> ⊥) -> ⊥:
  br(x = 0, b, z) where
    b() -> ⊥:
      c()
    z() -> ⊥:
      c()
    c() -> ⊥:
      ret(x)
```

(b) Classic CPS with new branch z

```
a(x: int, ret: cn(int)):
  b()
b():
  c()


c():
  ret(x)
```

(c) Thorin version

```
a(x: int, ret: cn(int)):
  br(x = 0, b, z)
b():
  c()
z():
  c()
c():
  ret(x)
```

(d) Thorin version with new branch z

Listing 9.3.: This example illustrates Thorin's blockless representation.

of CPS terms, the need to allocate closures for continuations, and the cumbersome nesting of continuations as opposed to a flat program structure used by CFG-based representations.

The disadvantage of nesting unveils in Listing 9.3. Assume we want to add a conditional branch in a to b and an additional continuation z that in turn branches to c. The original nesting must be repaired since c needs to be visible from both b and z (see Listing 9.3b). *Block/let floating* [San95] has to be applied to float c into a. The reverse transformation, i.e. sinking c into b, is known as *block sinking*. Such situations occur during several optimizations like, for example, jump threading[1].

9.1.3. A-Normal Form

administrative normal form (ANF) [Fla+93] is a direct-style program representation that still maintains the property of CPS programs to name all intermediate computations. Hence, ANF does not allow nested function

[1] see http://beza1e1.tuxen.de/articles/jump_threading.html

calls—unlike the usual λ-calculus. Thus, compared to CPS, terms in ANF are less complicated.

Example 9.6 (ANF)
This is the term from Example 9.4 in ANF:

```
let r_f = f(y) in
  let g = λ x. 23 in
    let r_g = g(r_f) in R
```

However, as Kennedy [Ken07] points out, in contrast to a faithful CPS representation ANF requires a re-normalization phase after β-reduction.

9.1.4. SSA Form vs. CPS

SSA form [RWZ88; Cyt+91] is a popular first-order program representation, which is typically used in compilers for imperative languages. This program representation is characterized by the fact that each variable is exactly defined once—just like in CPS. Consider the imperative program in Listing 9.4a, its translation into SSA form (see Listing 9.4b), as well as its CPS conversion (see Listing 9.4c). The CPS version acknowledges the fact that functions never return by using $\bot$ as return type.

SSA form introduces ϕ-functions in order to merge values from different predecessors (see the definitions of $\texttt{r}_1$ and $\texttt{i}_1$ in Listing 9.4b). The CPS version, however, introduces parameters for a continuation (parameters `i` and `r` in the continuation `head`). The arguments to the ϕ-function in the SSA-form version appear as arguments to a call of `head` in the CPS version.

The continuation `next` in Listing 9.4c makes use of `fac`'s higher-order parameter `ret`. This is legitimate as continuations may use parameters of other continuations in which they are nested. In the scope of the inner continuation these uses appear as free variables. Suppose `next` were not defined locally in `fac`. In that case, callers of `next` would have to pass `ret` as an additional parameter to `next`. To keep the number of parameters low, continuations are nested. Nesting continuations according to the dominance tree results in the minimal number of parameters [DS00, sec. 4.1].

As we have seen, differences between the SSA form and CPS are mostly syntactical. This has already been noted by Kelsey [Kel95] and Appel

```
fn fac(n: int) -> int {
  if n <= 1 {
    1
  } else {
    let mut r: int = 1;
    let mut i: int = 2;
    while i <= n {
      r *= i;
      ++i;
    }

    r
  }
}
```

(a) Original program

```
fn fac(n: int) -> int {
  br(n ≤ 1, then, else)
then:
  return 1;
else:
  int r0 ← 1;
  int i0 ← 2;
head:
  int r1 ← φ(r0 [else], r2 [body]);
  int i1 ← φ(i0 [else], i2 [body]);
  br(i1≤n, body, next)
body:
  int r2 ← r1 * i1;
  int i2 ← i1 + 1;
  goto head;
next:
  return r1;
}
```

(b) SSA-form version

Listing 9.4.: Factorial in different versions. In Thorin, each definition is a node. Every use is an edge to this node.

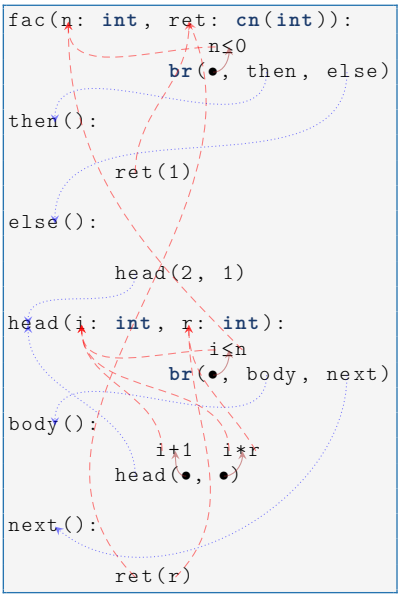
[App98]. To sum up: In contrast to SSA form, a CPS program can be higher-order. Furthermore, a CPS program requires continuations to be nested in order to disambiguate a variable use when the variable name is defined more than once in the program.

Graph-Based IRs

Classic SSA-based representations consist of a CFG and each basic block in turn consists of a rather rigid instruction list. Inspired by program dependence graphs (PDGs) [FOW87] and program dependence webs (PDWs) [BMO90], Click and Paleczny [CP95] promoted the idea of replacing these instruction lists with a data dependence graph. Each node in that graph represents an operation and each outgoing edge represents an operand of that operation (see Figure 9.1). Graph nodes also prevent name capture as discussed at the end of Section 9.1.1. The exact *placement* of a node—this means its associated basic block as well as its exact place within that block—is by design *not* recorded. Instead, a scheduler decides on the

```
fac(n: int, ret: int -> ⊥):
  br(n ≤ 1, then, else) where
    then():
      ret(1)
    else():
      head(2, 1) where
        head(i: int, r: int):
          br(i ≤ n, body,
             next) where
            body():
              head(i + 1, i * r)
            next():
              ret(r)
```

(c) Classic CPS version



(d) Thorin version (blockless)

Listing 9.4. (cont.)**:** Names are solely present for readability. They do not possess any semantic meaning.

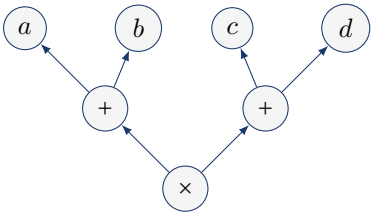


Figure 9.1.: Data dependence graph for Listing 9.2

placement of each node late during compilation. This corresponds to *code motion* in a classic IR. Click [Cli95] coined the term "sea of nodes" for this compiler design and argues that besides code motion several optimizations such as *(conditional) constant propagation*, *dead code elimination*, or *global value numbering* work well on a "sea of nodes". Other compilers such as FIRM [BBZ11], the Java HotSpot compiler [PVC01], and TurboFan[2] also employ a "sea of nodes".

Higher-Order Programs

A major restriction of SSA-form programs is that these programs can only be of first order. But many modern imperative programming languages like C++11, Java 8, Scala, Go, Rust, and Swift support higher-order functions. As already mentioned in Section 8.2.2, the common way to implement a function value is a closure. For example, the C++ code in Listing 9.5a results into the (stylized) code in Listing 9.5b. This transformation is called *closure-conversion* [App06, chap. 10]. However, a straightforward implementation of closures can incur a significant performance penalty: Ideally, the code in this example is compiled into a simple loop.

Since SSA-based IRs are too low-level to represent free variables directly, compilers already implement closure-conversion in the front end. This has several drawbacks:

- The implementation is language-specific and can hardly be reused in another front end.
- The IR code is significantly bloated. For every function abstraction a new `struct` is created. For example, the LLVM code for Listing 9.5b consists of over 600 lines.
- Finally, it is inelegant and inefficient to lower constructs that have to be restored later on. LLVM, for example, uses a combination of carefully coordinated analyses and transformations to eliminate closures: Inline the call to the closure's function pointer to be able to SSA-construct (`mem2reg`) the closure `struct` and finally dissolve the `struct` to scalar values. This strategy, for instance, fails to optimize recursive higher-order functions, like the one in the example above.

[2] see `https://github.com/v8/v8/wiki/TurboFan`

After optimizations, the example in Listing 9.5a is still more than 250 lines of LLVM bitcode long.

9.2. Thorin

9.2.1. Overview

Thorin blends concepts of a graph-based SSA-form and CPS and is equally well-suited to represent imperative as well as functional programs. Like SSA-form but unlike conventional functional IRs, Thorin has no scope nesting. Instead, Thorin embraces the "sea of nodes" paradigm and does not use named variables. In Thorin, each expression is a node in a graph and every reference to an expression is an edge to this node. Therefore, Thorin does not require explicit scope nesting. Listing 9.4d shows the Thorin graph for the factorial function: A Thorin program consists of a set of continuations. In Thorin, a continuation introduces parameters and in turn solely consists of a call. The callee and the arguments to that call reference other definitions: continuations (dotted edges), parameters (dashed edges), or primops (solid edges). A primop is a simple operation, which references other expressions to produce a new value—just like in Figure 9.1. Note that examples nevertheless use names in Thorin programs to make the presentation more accessible for humans. Names have no meaning otherwise.

In Thorin, the nesting is *implicitly* given by data dependencies between continuations. If a continuation f uses a variable defined in another continuation g, f is *implicitly* nested in g. Since Thorin does not have nesting, it abolishes block floating and sinking (see Listing 9.3c-d). We say Thorin is *blockless*.

Another reason to abandon nesting is that a given nesting is not necessarily minimal.

Example 9.7 (Let-Floating)
Suppose, a compiler wants to inline the call to `f` in the Listing 9.6a. The inliner would inline g as well although g does not use any of `f`'s parameters. This is why compilers for functional languages usually perform the aforementioned let-floating transformation beforehand [San95, chap. 3.4]. Now, inlining `f` does not touch g (see Listing 9.6b).

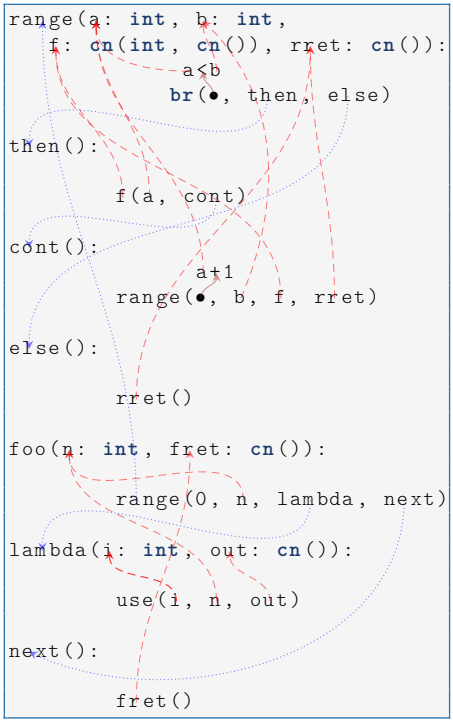
```
void range(int a, int b,
    function<void(int)> f) {
  if (a < b) {
    f(a);
    range(a+1, b, f);
  }
}

void foo(int n) {
  range(0, n, [=] (int i) {
    use(i, n);
  });
}
```

(a) Original C++ program

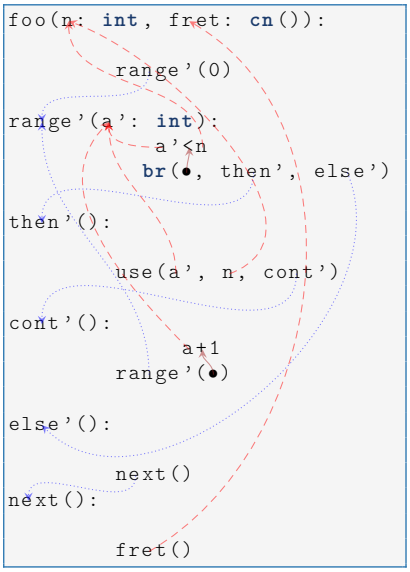
```
struct closbase {
  void (*f)(void* c, int i);
};

struct closure {
  closbase base;
  int n;
};

void lambda(void* c, int i) {
  use(i, (closure* c)->n);
}

void range(int a, int b,
           void* c) {
  if (a < b) {
    ((closbase*) c)->f(c, a);
    range(a+1, b, c);
  }
}

void foo(int n) {
  closure c = {{&lambda}, n};
  range(0, n, &c);
}
```

(b) Stylized imperative IR

Listing 9.5.: A higher-order function `range` and a call site within `foo`



(c) Thorin version



(d) Optimized Thorin version

Listing 9.5. (cont.)**:** the example in Thorin

```
let f a b =
    let g x =
        x+1
    in
    a + (g b)
in
    (f ...)
```

(a) g is nested in f

```
let g x =
    x+1
in
    let f a b =
        a + (g b)
    in
        (f ...)
```

(b) let-float g out of f

Listing 9.6.: Let-floating

Put another way, a classic compiler asks the question:

> Here is a function f. Are there any functions nested inside of f that I can block float outwards?

Whereas Thorin asks the question:

> Here is a function f. Which functions belong to f?

It is neither necessary to explicitly construct nesting when translating the source program to Thorin nor to repair nesting after code transformations. In fact, Impala exploits an SSA construction algorithm [Bra+13] to directly construct a Thorin program from the AST without further analyses like the computation of a dominance tree.

Let us come back to Listing 9.5a. The Thorin version (see Listing 9.5c) of that program is a straightforward translation of the source program because Thorin can directly represent free variables. In Chapter 11, we will discusses how to optimize this program (see Listing 9.5d) and generate code that does not need costly closures at runtime.

We argue that even classic compilers for imperative languages would profit from a higher-order IR. It is often important to annotate code regions (such as to mark a loop for parallelization). LLVM, for example, uses metadata to annotate code. This introduces many subtle problems. For instance, transformations must pay attention to not mistakenly destroy metadata designated for a different pass. Higher-order functions solve this problem in a clean and type-safe way by wrapping the code region to be annotated in a higher-order function (see Section 10.7.1).

9.2.2. Syntax

A Thorin *program* Φ (see Figure 9.2) is a set of *continuations*. A continuation consists of its label, signature, and *body* $b = e_0(\overline{e})$. This in turn is a call to e_0 with arguments $\overline{e}$. An *expression* is either

1. a *primop* $\boxtimes(\overline{e})$, where $\boxtimes$ is an operator (e.g. +, −, ...),
2. the abstraction of a continuation ℓ, or
3. a *parameter* x of a continuation.

> *Remark.* Values of type `int` or `bool` are just nullary primops. We use 42 and `true` as syntactic sugar for 42() and `true`(), respectively. We write ν_t to denote some value of type t. For example, the value $\nu_{\texttt{bool}}$ is either the primop `true`() or `false`().

Continuation labels range over ℓ, parameter names over x. We require both to be unique. Note that continuations may reference each other in a (possibly mutually) recursive way. Furthermore, mind that we use the label ℓ only in the formalism of Thorin to refer to the node that contains the continuation ℓ. In Thorin's implementation, labels do not exist. In the last section, we have already discussed how a Thorin graph looks like. However, for the remainder of this thesis, we waive edges and use unique names to resolve the declaration of a definition in order to increase readability.

9.2.3. The Scope

As outlined in previous sections, continuations in a Thorin program are not *explicitly* nested. For example, in Listing 9.4d the continuations `then` and `next` *directly* depend on `fac` as both continuations use `fac`'s parameter `ret`. Likewise, `head` *directly* depends on `fac` because `head` uses `fac`'s parameter `n`. In Listing 9.4d the continuation `else` depends on `fac` albeit `else`'s body does not directly use any of `fac`'s parameters. However, `else` invokes `head`, which directly depends on `fac`. Therefore, the continuation `else` *indirectly depends* on `fac` and, hence, is also *implicitly* nested in `fac`. For many analyses and transformations (including lambda mangling, as presented in Section 11.1), we need to know all direct and

$$
\begin{array}{rll}
t & ::= \texttt{bool} \mid \texttt{int} & \text{(type)} \\
& \mid \texttt{cn}(\overline{t}) & \text{(continuation type)} \\
\Phi & ::= \varnothing \mid \Phi \cup \{f\} & \text{(program)} \\
f & ::= \ell(\overline{x : t}) : b & \text{(continuation)} \\
b & ::= e(\overline{e}) & \text{(body)} \\
e & ::= & \text{(expression)} \\
& \quad \boxtimes\,(\overline{e}) & \text{(primop)} \\
& \mid \ell & \text{(abstraction)} \\
& \mid x & \text{(parameter)}
\end{array}
$$

Figure 9.2.: Syntax of Thorin

$$\boxed{\Phi \vdash \ell \text{ live } \ell'}$$

$$\text{L-Abs}\ \frac{\ell'(\cdots) : b' \in \Phi \qquad \ell'' \preceq b' \qquad \Phi \vdash \ell \text{ live } \ell''}{\Phi \vdash \ell \text{ live } \ell'}$$

$$\text{L-Param}\ \frac{\ell(\ldots, x : t, \ldots) : b \in \Phi \qquad \ell'(\cdots) : b' \in \Phi \qquad x \preceq b'}{\Phi \vdash \ell \text{ live } \ell'}$$

$$\text{L-Refl}\ \frac{}{\Phi \vdash \ell \text{ live } \ell} \qquad \text{L-Trans}\ \frac{\Phi \vdash \ell'' \text{ live } \ell' \qquad \Phi \vdash \ell' \text{ live } \ell}{\Phi \vdash \ell'' \text{ live } \ell}$$

Figure 9.3.: Liveness in Thorin

indirect dependencies from the view of a continuation. A liveness analysis (see Figure 9.3) obtains this information. It holds $\Phi \vdash \ell'$ live ℓ if continuation ℓ' is live in ℓ. This means ℓ might depend on ℓ'. L-Param finds direct dependencies, whereas L-Abs determines indirect dependencies.

Definition 9.1 (Liveness)
We call all continuations that are live from the view of an another continuation ℓ_e the *scope of* ℓ_e. We call ℓ_e the *entry continuation* of that scope:

$$scope_\Phi(\ell_e) \coloneqq \{\ell(\cdots) : b \mid \ell(\cdots) : b \in \Phi \ \wedge\ \Phi \vdash \ell_e \text{ live } \ell\}\ .$$

Remark. Although a scope is also just a program, we use the term scope in order to stress that the scope is a subset of the original program under consideration.

Definition 9.2 (Well-Formedness)
A program Φ is *well-formed* if (Φ, live) is a tree.

This definition implies that (Φ, live) must be a partially ordered set. Reflexivity and transitivity hold by definition. Antisymmetry means:

$$\text{If } \Phi \vdash \ell \text{ live } \ell' \text{ and } \Phi \vdash \ell' \text{ live } \ell, \text{ then } \ell = \ell'.$$

This condition is violated if two continuations reference a parameter of each other, and liveness becomes cyclic:

Example 9.8 (Ill-Formed Program: Antisymmetry (L-Param))
Consider the following program Φ:

```
f(x: int): g(y)
g(y: int): f(x)
```

The continuation `f` uses `g`'s parameter `y`. Conversely, the continuation `g` uses `f`'s parameter `x`. We can now derive:

$$\text{L-Param } \frac{\texttt{x} \preceq \texttt{f(x)}}{\Phi \vdash \texttt{f live g}} \qquad \text{L-Param } \frac{\texttt{y} \preceq \texttt{g(y)}}{\Phi \vdash \texttt{g live f}}$$

Hence, the program Φ is ill-formed because $\texttt{f} \neq \texttt{g}$. Of course, there can also be longer dependency chains that form a cycle.

This effect can be disguised by adding continuations as indirection:

Example 9.9 (Ill-Formed Program: Antisymmetry (L-Abs))
As a second example consider the following program Φ:

```
f(x: int): i()
g(y: int): h()
h():       a(x)
i():       a(y)
a(z: int): a(z)
```

We can now derive:

$$\text{L-Abs}\ \frac{\texttt{h} \preceq \texttt{h()} \qquad \text{L-Param}\ \dfrac{\texttt{x} \preceq \texttt{a(x)}}{\Phi \vdash \texttt{f live h}}}{\Phi \vdash \texttt{f live g}}$$

$$\text{L-Abs}\ \frac{\texttt{i} \preceq \texttt{i()} \qquad \text{L-Param}\ \dfrac{\texttt{y} \preceq \texttt{a(y)}}{\Phi \vdash \texttt{g live i}}}{\Phi \vdash \texttt{g live f}}$$

Hence, the program Φ is ill-formed because $\texttt{f} \neq \texttt{g}$.

Finally, (Φ, live) must form a tree. This is not the case in the following example, as liveness forms a directed acyclic graph (DAG):

Example 9.10 (Ill-Formed Program: DAG)
As a third example consider the following program Φ:

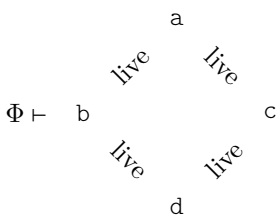
```
a(x: int): foo(b, c)
b(y: int): bar(x, d)
c(z: int): bar(x, d)
d(): ...y...z...
```

We can now derive:

$$\text{L-Param}\ \frac{\texttt{x} \preceq \texttt{bar(x, d)}}{\Phi \vdash \texttt{a live b}} \qquad \text{L-Param}\ \frac{\texttt{x} \preceq \texttt{bar(x, d)}}{\Phi \vdash \texttt{a live c}}$$

$$\text{L-Param}\ \frac{\texttt{y} \preceq \texttt{...y...z...}}{\Phi \vdash \texttt{b live d}} \qquad \text{L-Param}\ \frac{\texttt{z} \preceq \texttt{...y...z...}}{\Phi \vdash \texttt{c live d}}$$

This yields a DAG and not a tree:



Since liveness in well-formed Thorin programs induces a tree order for continuations, we can always reconstruct a nesting and translate such a program to λ^{cps}—a variant of Thorin with *explicit* nesting (see Section 9.3 and Example 9.11). This property is akin to the *strictness* property [Bud+02] in SSA-form programs:

> Each definition must dominate all of its uses.

However, this definition is actually "heavier" than ours: It requires a dominance analysis on a CFG—and a CFG is *not* part of Thorin's syntax and would have to be inferred (see Section 10.6). Our definition is more basic and avoids the notion of a CFG or dominance. It merely restricts the use of continuations and their parameters in a "strict" way.

Convention 9.1 (Well-Formedness)
From now on, we only consider well-formed Thorin programs.

9.3. λ^{cps}

The language λ^{cps} (see Figure 9.4) differs from Thorin in that nesting is explicitly given by the **where**-clause. The body b **where** $f_1, \dots, f_n$ binds f_1 to f_n in a mutually recursive way and makes these continuations visible in b. This is similar to Haskell's **whererec** or an upside-down version of **letrec** as in LISP/Scheme, ML, or other functional languages:

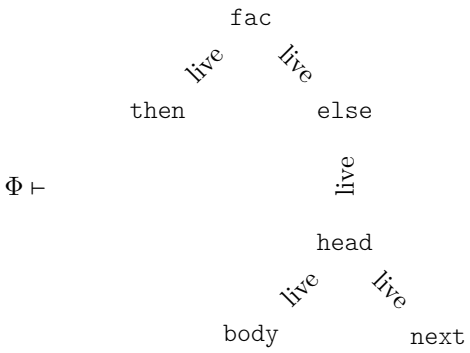
```
letrec f_1 ... f_n in b
```

For this reason, a λ^{cps} program is just a body b. In contrast to Thorin, there is no need to pool all continuations into a set. Additionally, λ^{cps} introduces closures as expanded syntax because expressions can reduce to continuations. Otherwise, λ^{cps} and Thorin are identical. Furthermore, we say *term* if we either mean an expression or a body.

λ^{cps} can also be seen as

- a CPS variant of Programming Computable Functions (PCF) [Plo77] because the **where**-clause allows (mutually) recursive functions,
- a higher-order variant of IL [SSH15], or
- a close relative of Appel's [App06] CPS language. However, Appel does not describe his language formally.

Example 9.11 (Reconstruct Nesting)
Reconsider the program Φ in Listing 9.4d. We can derive:



Using this tree, we reconstruct nesting and translate the Thorin- to the λ^{cps}-program in Listing 9.4c.

Of course, we can trivially translate a λ^{cps} program to Thorin by simply "forgetting" the nesting.

9.3.1. Typing

For the purpose of presentation, we restrict the type system of λ^{cps} to three types: the zeroth-order types `int` and `bool`, and the higher-order continuation type $\texttt{cn}(\overline{t})$. As continuations do not return, they do not possess a return type.

Definition 9.3 (Order)
The order of a type is defined as follows:

$$\begin{aligned} \mathit{order}(\texttt{bool}) &= 0 \\ \mathit{order}(\texttt{int}) &= 0 \\ \mathit{order}(\texttt{cn}(t_1, \dots, t_n)) &= 1 + \max(\mathit{order}(t_1), \dots, \mathit{order}(t_n)) \end{aligned}$$

Remark. Continuation types are at least of first order. Parameters can be of zeroth order or higher.

Example 9.12 (Order)
The type `cn(int,cn(bool))` denotes a type of a second-order continuation that expects two arguments: an integer and a first-order continuation that expects a boolean.

In both Thorin and λ^{cps} we use the continuation `br` of type

```
cn(bool, cn(), cn())
```

for conditional branches. Furthermore, invoking `exit`(ν) of type `cn(int)` ends evaluation with the result ν.

Definition 9.4 (Exits)
Let $\mathit{exits}(b) \coloneqq \{\texttt{exit}\ e \,|\, \texttt{exit}\ e \preceq b\}$ be the set of exits in the body b.

The type system of λ^{cps} is presented in Figure 9.5. Note that body rules do not yield a type in contrast to expression rules since calling continuations never returns. T-Where checks all bound continuations and the body of a `where`-clause by putting all bound continuations into the typing environment. T-App checks all arguments and whether the type of the callee e_0 actually fits to the argument list. T-Abs resolves the type of a continuation by looking up its signature in the typing environment. T-Param infers the type of a parameter by projecting from its continuation's signature the corresponding type. A primop only expects specific input types and in turn has a specific output type. The function $\mathit{check}_{\boxtimes}(\overline{e})$ in T-Primop handles these primop-specific rules. T-Clos puts the closure's parameters along with their associated types into the typing environment and checks the closure's body.

Definition 9.5 (Well-Typedness)
We call a body b *well-typed* under Γ iff $\Gamma \vdash b$ holds.

Remark. Note that $\vdash b$ implies b not having free variables (see below).

$$
\begin{array}{rll}
\Gamma & ::= & \varnothing \mid \Gamma, v : t & \text{(typing environment)} \\
t & ::= & \texttt{bool} \mid \texttt{int} & \text{(type)} \\
& \mid & \texttt{cn}(\overline{t}) & \text{(continuation type)} \\
f & ::= & \ell(\overline{x : t}) : b & \text{(continuation)} \\
b & ::= & & \text{(body)} \\
& & e(\overline{e}) & \text{(application)} \\
& \mid & b \ \texttt{where} \ \overline{f} & \text{(where)} \\
e & ::= & & \text{(expression)} \\
& & \boxtimes\,(\overline{e}) & \text{(primop)} \\
& \mid & \ell & \text{(abstraction)} \\
& \mid & x & \text{(parameter)} \\
& \mid & f & \text{(closure)} \\
v & ::= & \ell \mid x & \text{(variable)}
\end{array}
$$

Figure 9.4.: Syntax of λ^{cps}. Expanded syntax is grayed out.

Definition 9.6 (Value)
We call an expression e a *value* iff e is a normal form and $\vdash e : t$ for some type t.

> *Remark.* These are all primop values and continuations without free variables (see below).

9.3.2. Reduction

In this section we are working towards a reduction system for λ^{cps}. Before delving into the reduction relation, we first have to discuss *free variables* and *substitution*.

Definition 9.7 (Free Variables)
The set of of free variables in a λ^{cps} term constitutes all variables that occur in this term and are not bound by any continuation/`where`-clause:

Body: $\boxed{\Gamma \vdash b}$

$$\text{T-Where}\ \frac{\Gamma' \coloneqq \Gamma, \ell_1 : \mathtt{cn}(\overline{t^1}), \ldots, \ell_n : \mathtt{cn}(\overline{t^n}) \qquad \Gamma' \vdash b_0 \qquad \Gamma', \overline{x^1 : t^1} \vdash b_1 \quad \cdots \quad \Gamma', \overline{x^n : t^n} \vdash b_n}{\Gamma \vdash b_0\ \mathtt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}$$

$$\text{T-App}\ \frac{\Gamma \vdash e_0 : \mathtt{cn}(t_1, \ldots, t_n) \qquad \Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash e_0(e_1, \ldots, e_n)}$$

Expression: $\boxed{\Gamma \vdash e : t}$

$$\text{T-Abs}\ \frac{\ell : \mathtt{cn}(\bar{t}) \in \Gamma}{\Gamma \vdash \ell : \mathtt{cn}(\bar{t})} \qquad \text{T-Param}\ \frac{x : t \in \Gamma}{\Gamma \vdash x : t} \qquad \text{T-Clos}\ \frac{\Gamma, \overline{x : t} \vdash b}{\Gamma \vdash (\ell(\overline{x : t}) : b) : \mathtt{cn}(\bar{t})}$$

$$\text{T-Primop}\ \frac{t = \mathit{check}_{\boxtimes,\Gamma}(\bar{e})}{\Gamma \vdash \boxtimes(\bar{e}) : t}$$

Figure 9.5.: Typing in λ^{cps}

$$\begin{aligned} FV(v) &\coloneqq \{v\} && \text{remember that } v ::= \ell \mid x \\ FV(e_0(e_1, \ldots, e_n)) &\coloneqq FV(e_0) \cup \cdots \cup FV(e_n) \\ FV(\ell(\overline{x : t}) : b) &\coloneqq FV(b) \setminus \{\overline{x}\} \\ FV(\boxtimes(e_1, \ldots, e_n)) &\coloneqq FV(e_1) \cup \cdots \cup FV(e_n) \end{aligned}$$

$$\begin{aligned} &FV(b\ \mathtt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n) \coloneqq \\ &\qquad FV(b) \cup FV(\ell_1(\overline{x^1 : t^1}) : b_1) \cup \cdots \cup FV(\ell_n(\overline{x^n : t^n}) : b_n) \setminus \{\ell_1, \ldots, \ell_n\} \end{aligned}$$

Definition 9.8 (Substitution)
Up to renaming of bound variables, substitution is defined as follows:

$$\begin{aligned} [v \mapsto e_s]v &\coloneqq e_s \\ [w \mapsto e_s]v &\coloneqq v && \text{if } v \neq w \\ [v \mapsto e_s](e_0(e_1, \ldots, e_n)) &\coloneqq [v \mapsto e_s]e_0([v \mapsto e_s]e_1, \ldots, [v \mapsto e_s]e_n) \\ [v \mapsto e_s](\ell(\overline{x : t}) : b) &\coloneqq \ell(\overline{x : t})[v \mapsto e_s]b && \text{if } v \notin \overline{x} \text{ and } \overline{x} \notin FV(e_s) \\ [v \mapsto e_s] \boxtimes (e_1, \ldots, e_n) &\coloneqq \boxtimes([v \mapsto e_s]e_1, \ldots, [v \mapsto e_s]e_n) \end{aligned}$$

$\boxed{b \to b'}$

$$\text{R-Cong}_{1a}\ \frac{b_0 \to b_0'}{b_0\ \texttt{where}\ \overline{f} \to b_0'\ \texttt{where}\ \overline{f}} \qquad \text{R-Cong}_{2a}\ \frac{e_0 \to e_0'}{e_0(\overline{e}) \to e_0'(\overline{e})}$$

$$\text{R-Cong}_{1b}\ \frac{b_i \to b_i' \qquad 1 \le i \le m}{\begin{array}{l} b_0\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \dots, \ell_i(\overline{x^i : t^i}) : b_i, \dots, \ell_n(\overline{x^n : t^n}) : b_n \\ \to b_0\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \dots, \ell_i(\overline{x^i : t^i}) : b_i', \dots, \ell_n(\overline{x^n : t^n}) : b_n \end{array}}$$

$$\text{R-Cong}_{2b}\ \frac{e_i \to e_i'}{e_0(e_1, \dots, e_i, \dots, e_n) \to e_0(e_1, \dots, e_i', \dots, e_n)}$$

$$\text{R-Where}\ \begin{array}{l} b_0\ \texttt{where}\ \overbrace{\ell_1(\overline{x^1 : t^1}) : b_1, \dots, \ell_n(\overline{x^n : t^n}) : b_n}^{\overline{f}} \\ \to [\ell_1 \mapsto \ell_1(\overline{x^1 : t^1}) : b_1\ \texttt{where}\ \overline{f}] \cdots [\ell_n \mapsto \ell_n(\overline{x^n : t^n}) : b_n\ \texttt{where}\ \overline{f}] b_0 \end{array}$$

$$\text{R-App}\ (\ell(x_1 : t_1, \dots, x_n : t_n) : b)(e_1, \dots, e_n) \to [x_1 \mapsto e_1] \cdots [x_n \mapsto e_n] b$$

$\boxed{e \to e'}$

$$\text{R-Primop}\ \frac{e_i \to e_i'}{\boxtimes(\dots, e_i, \dots) \to \boxtimes(\dots, e_i', \dots)} \qquad \text{R-Fold}\ \frac{n > 0}{\boxtimes(\nu_{t1}^1, \dots, \nu_{tn}^n) \to \otimes(\nu_{t1}^1, \dots, \nu_{tn}^n)}$$

$$\text{R-Clos}\ \frac{b \to b'}{(\ell(\overline{t}) : b) \to (\ell(\overline{t}) : b')}$$

Figure 9.6.: Reduction ($\to$) in λ^{cps}

$$[v \mapsto e_s](b\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \dots, \ell_n(\overline{x^n : t^n}) : b_n) \coloneqq$$
$$[v \mapsto e_s]b\ \texttt{where}\ [v \mapsto e_s](\ell_1(\overline{x^1 : t^1}) : b_1), \dots, [v \mapsto e_s](\ell_n(\overline{x^n : t^n}) : b_n)$$
$$\text{if } v \notin \ell_1, \dots, \ell_n \text{ and } \ell_1, \dots, \ell_n \notin FV(e_s)$$

Remark. At first glance, the guards in the definition of FV seem to be non-exhaustive. However, since this definition is "up to renaming of bound variables", we α-convert terms until the required guards are met.

Example 9.13 (α-conversion)
Since $x \in \{x\}$, the guard for the case of a closure is not met in the following substitution: $[x \mapsto e_s](f(x : t) : b)$. After α-conversion, we

$\boxed{b \twoheadrightarrow b'}$

$$\text{P-Cong}_1 \ \frac{b_0 \twoheadrightarrow b_0' \quad \cdots \quad b_n \twoheadrightarrow b_n'}{b_0 \ \texttt{where}\ \ell_1(\overline{t}_1) : b_1, \ldots, \ell_n(\overline{t}_n) : b_n \twoheadrightarrow b_0' \ \texttt{where}\ \ell_1(\overline{t}_1) : b_1', \ldots, \ell_n(\overline{t}_n) : b_n'}$$

$$\text{P-Cong}_2 \ \frac{e_0 \twoheadrightarrow e_0' \quad \cdots \quad e_n \twoheadrightarrow e_n'}{e_0(e_1, \ldots, e_n) \twoheadrightarrow e_0'(e_1', \ldots, e_n')}$$

$$\text{P-Where} \ \frac{b_0 \twoheadrightarrow b_0' \quad \cdots \quad b_n \twoheadrightarrow b_n'}{\begin{array}{l} b_0 \ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n \\ \twoheadrightarrow \cdots[\ell_i \mapsto \ell_i(\overline{x^i : t^i}) : b_i' \ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1', \ldots, \ell_n(\overline{x^n : t^n}) : b_n'] \cdots b_0' \end{array}}$$

$$\text{P-App} \ \frac{b \twoheadrightarrow b' \qquad e_1 \twoheadrightarrow e_1' \quad \cdots \quad e_n \twoheadrightarrow e_n'}{(\ell(x_1 : t_1, \ldots, x_n : t_n) : b)(e_1, \ldots, e_n) \twoheadrightarrow [x_1 \mapsto e_1'] \cdots [x_n \mapsto e_n'] b'}$$

$\boxed{e \twoheadrightarrow e'}$

$$\text{P-Abs} \ \frac{}{\ell \twoheadrightarrow \ell} \qquad \text{P-Param} \ \frac{}{x \twoheadrightarrow x} \qquad \text{P-Primop} \ \frac{e_1 \twoheadrightarrow e_1' \qquad \cdots \qquad e_n \twoheadrightarrow e'n}{\boxtimes(e_1, \ldots, e_n) \twoheadrightarrow \boxtimes(e_1', \ldots, e_n')}$$

$$\text{P-Fold} \ \frac{}{\boxtimes(\nu_{t1}^1, \ldots, \nu_{tn}^n) \twoheadrightarrow \otimes(\nu_{t1}^1, \ldots, \nu_{tn}^n)} \qquad \text{P-Clos} \ \frac{b \twoheadrightarrow b'}{(\ell(\overline{x : t}) : b) \twoheadrightarrow (\ell(\overline{x : t}) : b')}$$

Figure 9.7.: Parallel reduction ($\twoheadrightarrow$) in λ^{cps}

obtain: $[x \mapsto e_s](f(y : t) : b)$. Now the guard is satisfied and we can proceed.

Substitution preserves well-typedness as the following lemma states.

Lemma 9.1 (λ^{cps}: Substitution – Typing)

$$\textit{If} \ \begin{array}{c} \Gamma \vdash \hat{b} \\ \Gamma \vdash \hat{e} : \hat{t} \end{array}, \Gamma \vdash v : t, \textit{ and } \Gamma \vdash e : t, \textit{ then } \begin{array}{c} \Gamma \vdash [v \mapsto e]\hat{b} \\ \Gamma \vdash [v \mapsto e]\hat{e} : \hat{t} \end{array} .$$

Proof sketch. By mutual induction on a derivation of $\begin{array}{c} \Gamma \vdash \hat{b} \\ \Gamma \vdash \hat{e} : \hat{t} \end{array}$.
See Section B.3 for the full proof.

The relation $\rightarrow$ (see Figure 9.6) defines a reduction system for λ^{cps}. Rules R-Cong$_{1a}$ and R-Cong$_{1b}$ allow us to arbitrarily descend into a `where`-clause,

whereas rules R-Cong$_{2a}$ and R-Cong$_{2b}$ allow us to arbitrarily descend into an application. R-Where substitutes all recursively bound functions into themselves and into the `where`-clause's body b. R-App reduces a closure in callee position to its body while substituting all parameters with their associated arguments. R-Primop allows us to arbitrarily descend into any of the primop's operand. R-Fold folds a primop that just consists of values; the symbol $\boxtimes$ is meant purely syntactically, while $\otimes$ refers to the actual mathematical operations like actually adding two values. Finally, R-Clos reduces the body of a closure.

Remark. The reduction rules are *indeterministic*. One is free to choose

- R-Cong$_{1a}$/R-Cong$_{1b}$ over R-Where,
- R-Cong$_{2a}$/R-Cong$_{2b}$ over R-App,
- any operand in R-Primop, or
- R-Fold instead of R-Primop if all operands are values.

Remark. Actually, Figure 9.6 defines two relations in a mutually recursive way. The relation $e \to e'$ uses $b \to b'$ and vice versa. However, for the sake of readability, we will often pretend that $\to$ is just one relation in the case that definitions, statements and so forth in fact hold for both relations alike.

We assume that primops are total. Lemma 9.2 establishes type preservation for total primops. In practice, however, not every primop meets this assumption. For instance, division by zero is not defined. Moreover, primops that cause side-effects are modeled with functional loads and stores [Ste95; Str00]. Dereferencing a dangling pointer ("use after free") is another example of a partial primop. As a consequence of Lemma 9.2, we know that the loss of preservation that we experience when we move to partial primops stems indeed from them and not from a foul design of continuations.

Lemma 9.2 (λ^{cps}: $\rightarrow$-Preservation)
If a well-typed λ^{cps} term reduces with $\rightarrow$, the resulting term is also well-typed. To be more precise:

$$\textit{If} \quad \begin{matrix} \Gamma \vdash b \\ \Gamma \vdash e : t \end{matrix} \quad \textit{and} \quad \begin{matrix} b \rightarrow b' \\ e \rightarrow e' \end{matrix} \ , \ \textit{then} \quad \begin{matrix} \Gamma \vdash b' \\ \Gamma \vdash e' : t \end{matrix} \quad .$$

Proof sketch. By mutual induction on a derivation of $\begin{matrix} b \rightarrow b' \\ e \rightarrow e' \end{matrix}$ and Lemma 9.1. See Section B.3 for the full proof.

9.3.3. Confluence

This section proves confluence of $\rightarrow$. The proof is based upon the technique of Tait/Martin-Löf as, for instance, presented by Barendregt [Bar84, Chapter 3]. We strongly recommend readers not familiar with this proof technique to first study it in the context of confluence for β-reduction in the untyped lambda calculus. In addition to Barendregt [Bar84], Pollack [Pol95], Selinger [Sel08, Chapter 4], and Smolka [Smo15], for example, discuss the proof thoroughly.

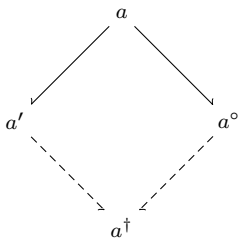
Definition 9.9 (Diamond Property)
Let $\rightharpoonup$ be a binary relation on a set A. We say $\rightharpoonup$ satisfies the *diamond property* (notation $dp(\rightharpoonup)$) if holds:
Whenever

$$a \rightharpoonup a' \quad \text{and} \quad a \rightharpoonup a^{\circ}$$

then there exists an $a^{\dagger} \in A$ such that

$$a' \rightharpoonup a^{\dagger} \quad \text{and} \quad a^{\circ} \rightharpoonup a^{\dagger} \ .$$

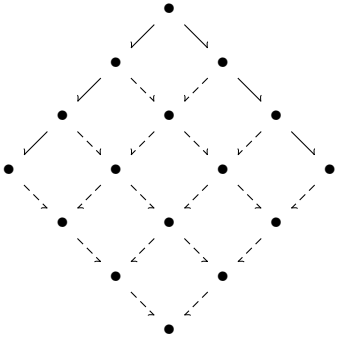
We graphically represent this property in the following picture:



Lemma 9.3 (Strip Lemma)
Let $\rightharpoonup$ be a binary relation and $\rightharpoonup^*$ its reflexive transitive closure.

$$\textit{If } dp(\rightharpoonup), \textit{ then } dp(\rightharpoonup^*) \ .$$

Proof. By a simple diagram chase as the following figure suggests:



□

> *Remark.* Pollack [Pol95, Section 2] gives a more technical proof.

Unfortunately, $\rightarrow$ does *not* satisfy the diamond property. Thus, we cannot simply apply Lemma 9.3 on $\rightarrow$ in order to prove $dp(\rightarrow^*)$. There are two reasons why $dp(\rightarrow)$ does not hold [cf. Pol95, Section 1.2]: Relation $\rightarrow$ can

1. "forget" subterms but is not reflexive, and
2. copy subterms but is not parallel.

The parallel reduction $\twoheadrightarrow$ (see Figure 9.7) fixes these issues. This relation's reflexive transitive closure is the same as the reflexive transitive closure of $\rightarrow$ (see Lemma 9.5). So, if we prove $dp(\twoheadrightarrow)$ (see Lemma 9.8), we also know $dp(\rightarrow^*)$ (see Theorem 9.1).

Lemma 9.4 (Reflexivity of $\twoheadrightarrow$)

$$\textit{For all } \begin{matrix} b \\ e \end{matrix} \textit{ holds } \begin{matrix} b \twoheadrightarrow b \\ e \twoheadrightarrow e \end{matrix} \ .$$

Proof. As base we have P-Abs, P-Param, and P-Fold with $n = 0$. When reducing a body, we can always choose between P-Cong, P-App, and P-Where. So let us choose P-Cong. When reducing a primop we can always choose between P-Fold and P-Primop. So let us choose P-Primop. By a straightforward induction we show reflexivity for the remaining cases. □

Lemma 9.5 ($\to^* = \twoheadrightarrow^*$)

(a) Whenever $\begin{matrix} b \to b' \\ e \to e' \end{matrix}$ *, then* $\begin{matrix} b \twoheadrightarrow b' \\ e \twoheadrightarrow e' \end{matrix}$.

(b) Whenever $\begin{matrix} b \twoheadrightarrow b' \\ e \twoheadrightarrow e' \end{matrix}$ *, then* $\begin{matrix} b \to^* b' \\ e \to^* e' \end{matrix}$.

(c) $\to^*$ *is the reflexive transitive closure of* $\twoheadrightarrow$.

Proof sketch.

(a) By mutual induction on a derivation of $\begin{matrix} b \to b' \\ e \to e' \end{matrix}$ and Lemma 9.4.

(b) By mutual induction on a derivation of $\begin{matrix} b \twoheadrightarrow b' \\ e \twoheadrightarrow e' \end{matrix}$ and transitivity of $\to^*$.

(c)
- By (a) we have $\to \subseteq \twoheadrightarrow$, hence $\to^* \subseteq \twoheadrightarrow^*$.
- By (b) we have $\twoheadrightarrow \subseteq \to^*$, hence $\twoheadrightarrow^* \subseteq \to^*$.
- Thus, $\to^* = \twoheadrightarrow^*$.

See Section B.3 for the full proof.

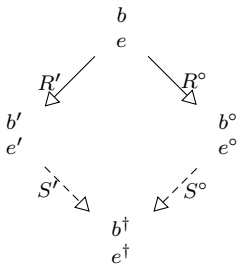
Lemma 9.6 (λ^{cps}: Substitution – Reduction)

$$\textit{If} \begin{matrix} \hat{b} \twoheadrightarrow \hat{b}' \\ \hat{e} \twoheadrightarrow \hat{e}' \end{matrix} \textit{ and } e \twoheadrightarrow e' \textit{ then } [v \mapsto e] \begin{matrix} \hat{b} \\ \hat{e} \end{matrix} \twoheadrightarrow [v \mapsto e'] \begin{matrix} \hat{b}' \\ \hat{e}' \end{matrix} .$$

Proof sketch. By mutual induction on a derivation of $\begin{matrix} \hat{b} \twoheadrightarrow \hat{b}' \\ \hat{e} \twoheadrightarrow \hat{e}' \end{matrix}$.

See Section B.3 for the full proof.

Finally, we have to prove the diamond property for $\twoheadrightarrow$. A direct proof requires a complicated double induction on R' and R°:



For all combinations of possible rules for R' and R° we have to find rules S' and S° to close the diamond. This requires us to consider a quadratic number of cases. Pfenning [Pfe92], for example, gives a direct proof in the context of the untyped lambda calculus. However, the *maximal parallel one-step reduct* [Tak95] offers a more elegant way:

Definition 9.10 (Maximal parallel one-step reduct)
The *maximal parallel one-step reduct* is defined as follows:

$$
\begin{aligned}
\rho[e_0(e_1,\dots,e_n)] &\coloneqq \rho[e_0]\,(\rho[e_1],\dots,\rho[e_n]) && \text{if } e_0 \text{ is not a closure}\\
\rho[(\ell(x_1:t_1,\dots,x_n:t_n):b)(e_1,\dots,e_n)] &\coloneqq [x_1 \mapsto \rho[e_1]]\cdots[x_n \mapsto \rho[e_n]]\rho[b] && \text{otherwise}\\
\rho[\ell] &\coloneqq \ell\\
\rho[x] &\coloneqq x\\
\rho\big[\ell(\overline{x:t}):b\big] &\coloneqq \ell(\overline{x:t}):\rho[b]\\
\rho[\boxtimes(e_1,\dots,e_n)] &\coloneqq \boxtimes(\rho[e_1],\dots,\rho[e_n]) && \text{if } e_1,\dots,e_n \text{ are not values}\\
\rho\big[\boxtimes(\nu^1_{t1},\dots,\nu^n_{tn})\big] &\coloneqq \otimes(\nu^1_{t1},\dots,\nu^n_{tn}) && \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
&\rho\big[b_0 \texttt{ where } \overline{f}\big] \coloneqq\\
&\quad\cdots\Big[\ell_i \mapsto \ell_i(\overline{x^i:t^i}):\rho[b_i] \texttt{ where } \ell_1(\overline{x^1:t^1}):\rho[b_1],\dots,\ell_n(\overline{x^n:t^n}):\rho[b_n]\Big]\cdots\rho[b_0]\\
&\qquad\qquad \text{if } \big|\overline{f}\big| \geq 1 \text{ where } \overline{f} \coloneqq \ell_1(\overline{t}_1):b_1,\dots,\ell_n(\overline{t}_n):b_n
\end{aligned}
$$

Remark. In contrast to the relations $\to$ and $\twoheadrightarrow$, the function ρ is deterministic due to the case distinctions in its definition.

Intuitively, the function ρ recursively disassembles a body/an expression, reduces closure applications and `where`-bindings, folds primops whenever

possible, and reassembles everything again. But since ρ *first* recurses into subterms and *then* reduces closure applications/`where`-bindings and folds primops, it will reduce each closure application and `where`-binding/fold each primop exactly *once*. For this reason, the maximal parallel one-step reduct is also sometimes called *complete development*. The following lemma implies the diamond property for $\twoheadrightarrow$. What is more, the proof only requires a linear number of cases to consider.

Lemma 9.7 (λ^{cps}: Maximal parallel one-step reductions)

$$\textit{Whenever} \quad \begin{matrix} b \twoheadrightarrow \hat{b} \\ e \twoheadrightarrow \hat{e} \end{matrix} \;, \; \textit{then} \quad \begin{matrix} \hat{b} \twoheadrightarrow \rho[b] \\ \hat{e} \twoheadrightarrow \rho[e] \end{matrix} \quad .$$

Proof sketch. By mutual induction on a derivation of $\begin{matrix} b \twoheadrightarrow \hat{b} \\ e \twoheadrightarrow \hat{e} \end{matrix}$ and Lemma 9.6. See Section B.3 for the full proof.

Lemma 9.8 (Diamond property for $\twoheadrightarrow$)
The relation $\twoheadrightarrow$ satisfies the diamond property.

Proof. Take $\begin{matrix} b^\dagger = \rho[b] \\ e^\dagger = \rho[e] \end{matrix}$ with Lemma 9.7. □

Theorem 9.1 (Confluence)
The relation $\twoheadrightarrow^*$ satisfies the diamond property.

Proof.

- By Lemma 9.8 we know $dp(\twoheadrightarrow)$.
- Lemma 9.3 implies $dp(\twoheadrightarrow^*)$.
- Thus, by Lemma 9.5 we conclude $dp(\twoheadrightarrow^*)$.

□

> *Remark.* This property is equivalent to the Church–Rosser property [CR36].

He who controls the past controls the future.
He who controls the present controls the past.

George Orwell, 1984

10 Partial Evaluation

The last chapter presents an *indeterministic* reduction system for λ^{cps}. However, an interpreter or compiler for λ^{cps} needs an *evaluation strategy*, i.e., a *deterministic* reduction system. This is why this chapter introduces a new relation $\Rightarrow \subset \rightarrow$: a call-by-value evaluation strategy. We will enhance $\Rightarrow$ to work on programs with free variables in a deterministic way. This allows us to use $\Rightarrow$ for both *full* as well as *partial evaluation*. Equipped with these insights, we will discuss how to embed and guide the partial evaluator from within a program and how this affects termination. Then, we take a look on how we can partially evaluate data structures. A crucial aspect of our partial evaluation algorithm is a higher-order notion of *post-dominators*. This requires a *control-flow analysis (CFA)*. Finally, our experiments demonstrate that the presented partial evaluation approach is able to remove the overhead of two shallowly embedded high-performance DSLs. But first, let us informally introduce our partial evaluation algorithm and discuss related work.

10.1. Overview

The partial evaluator works as follows:

1. If the callee of a function application is known, unfold this call and proceed.

2. Otherwise, skip to a post-dominator of that call and proceed evaluation there.

```
fn count(i: int, N: int) -> int {
    if i<N {
        count(i+1, N)
    } else {
        i
    }
}

fn loops(a: int, b: int) -> int {
    let x = count(0, a);
    let y = count(0, b);
    x + y
}

let z = @loops(3, D)
```

Listing 10.1.: Two instances of the counting-loop problem

Example 10.1 (Partial Evaluation)
Consider function `count` in Listing 10.1. It consists of two counting loops—two instances of the *counting loop problem* (see Section 10.2.4). Using our partial evaluation strategy, we evaluate the program as depicted in Listing 10.2. Each step refers to one step in that figure.

(a) After having inlined `loops`, the partial evaluator inlines `count`.

(b) The evaluator folds the condition `0<3` and keeps the call `count(1, 3)`.

(c) The evaluator keeps inlining `count` until ...

(d) ... the condition `3<3` evaluates to false.

(e) Thus, `3` is propagated for `x` and the evaluator inlines `count` for `y`.

(f) As the condition `0<D` is dynamic the evaluator continues at the post-dominator ...

(g) ... and the partial evaluator terminates with the final residual program.

```
let z = {
  let x = count(0, 3);




  let y = count(0, D);
  x + y
}
```

(a)

```
let z = {
  let x = if 0<3 {
            count(1, 3)
          } else {
            0
          };
  let y = count(0, D);
  x + y
}
```

(b)

```
let z = {
  let x = count(1, 3);




  let y = count(0, D);
  x + y
}
```

(c)

```
let z = {
  let x = if 3<3 {
            count(4, 3)
          } else {
            3
          };
  let y = count(0, D);
  x + y
}
```

(d)

```
let z = {
  let y = count(0, D);




  3 + y
}
```

(e)

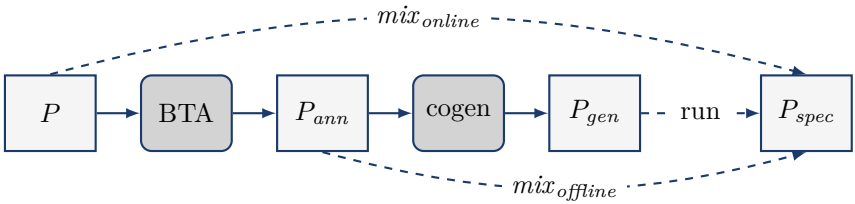
```
let z = {
  let y = if 0<D {
            count(1, D)
          } else {
            0
          };
  3 + y
}
```

(f)

```
let z = 3 + if 0<D { count(1, D) } else { 0 };
```

(g)

Listing 10.2.: Step-by-step example of the partial evaluation algorithm




```
fn Pann(x:int, n:int) -> int {
  if n == 0 {
    1
  } else if n % 2 == 0 {
    let r = Pann(x, n/2)
    r * r
  } else {
    x * Pann(x, n-1)
  }
}
```

(a) P_{ann}

```
fn Pspec(x:int) -> int {
  let r = x*x;
  r*r
}
```

(b) P_{spec} with `Pgen(4)`

```
fn Pgen'(n:int) -> code {
  if n == 0 {
    "1"
  } else if n % 2 == 0 {
    let r:code = Pgen'(n/2);
    r + "*" + r
  } else {
    "x*" + Pgen'(n-1)
  }
}

fn Pgen(n:int) -> () {
  emit(
    "fn Pspec(x:int) -> int {"
  );
  emit(Pgen'(n));
  emit("}");
}
```

(c) P_{gen}

Listing 10.3.: Partial evaluation and metaprogramming

10.2. Related Work

Program specialization goes back to Kleene's [Kle38] s_n^m-theorem. It states that there exists a particular algorithm that accepts m values and a program with $m+n$ free variables. This algorithm generates a program that only has n free variables by substituting the m values for the m free variables.

As running example to discuss prior work, we review how to specialize the power function to its exponent.

```
let rec pow x n =
  if n = 0 then
    .<1>.
  else if even n then
    let r = pow x (n/2) in
    r * r
  else
    .<.~x *
      .~(pow x (n-1))>.


let Pspec =
  .<fun x ->
    .~(pow .<x>. 4)>.
```

(a) MetaOCaml

```
function pow(x, n)
  if n == 0 then
    return 1
  elseif n % 2 == 0 then
    local r = pow(x, n/2)
    return `[r]*[r]
  else
    return `[x]*[pow(x, n-1)]
  end
end

terra Pspec(y: int)
  return [pow(y, 4)]
end
```

(b) Terra

```
trait Pow { this: Arith =>
  def pow(x:Rep[Int], n:Int):
      Rep[Int]= {
    if (n == 0) {
      1
    } else if (n % 2 == 0) {
      val r = pow(x, n/2);
      r * r
    } else {
      x * pow(x, n-1)
    }
  }
}
//...
val o = new Pow with ArithExp
import o._
val Pspec = pow(fresh[Int],4)
```

(c) Scala/LMS

```
fn pow(x: int, n: int) -> int {
  if n == 0 {
    1
  } else if n % 2 {
    let r = pow(x, n/2);
    r * r
  } else {
    x * pow(x, n-1)
  }
}



fn Pspec(y: int) -> int {
  @pow(y, 4)
}
```

(d) Impala

Listing 10.4.: Specializing the power function to its exponent with metaprogramming and partial evaluation. Code needed to achieve this specialization is highlighted.

10.2.1. Partial Evaluation

In this thesis, we advocate *online* partial evaluation [Lom67; Ruf93; CL11; SC11]: We directly specialize the source program P (see Listing 10.4d) on the fly without prior analysis to the *specialized* program or *residuuum* P_{spec} (see Listing 10.3b). This corresponds to the first Futamura [Fut99] projection: Specializing an interpreter P to an input program produces a compiled version of that program. The specializer is often called *mix* in literature.

Specializing the specializer for itself yields a *compiler generator (cogen)* or *generating extension generator (gegen)*: a program that converts an interpreter to an equivalent compiler (the third [Fut99] projection). In order to actually achieve this, the specializer must be self-applicable—a requirement which is hard to realize in practice [Glü12]. This is much easier for an *offline* [JSS89; Jon95; BJ93] evaluator: First, a *binding-time analysis (BTA)* [JS86; JSS89] identifies which parts of the program can be static and which ones must remain dynamic. The result is P_{ann} (see Listing 10.3a)—a program in which the *binding-time*, i.e. static or dynamic, is *annotated*. Then, the specializer ($mix_{offline}$) runs on that annotated program as opposed to directly running the specializer (mix_{online}) on P. Birkedal and Welinder [BW94] discovered that hand-writing cogen is actually not more difficult than writing an ordinary offline evaluator. In particular, a hand-written cogen does not require a bootstrapping process. This has the additional advantage that cogen's output language can be different from its input language while cogen itself may even be written in a third language. Given the annotated program P_{ann}, cogen produces its *generating extension* P_{gen} (see Listing 10.3c): All static parts of P_{ann} are copied over to P_{gen}. Dynamic parts are converted into a program that generates the specialized program P_{spec}. Thus, P_{gen} is parametric in P_{ann}'s static input. Running P_{gen} with a specific static input generates a program P_{spec}, which is parametric in P_{ann}'s dynamic input. For example, invoking `Pgen(4)` *generates* P_{spec} (see Listing 10.3b).[1] From a different point of view, cogen transforms the one-stage program P into a two-stage program. Moreover, once cogen's compiler has been generated, the performance of running this generator is usually much better than running an online evaluator.

[1] This listing assumes that the code generator that emits `Pspec` is smart enough to also fold `x * 1` to `x`.

On the downside, a BTA analysis is inherently less precise than a sophisticated online evaluator because a BTA only differentiates between static and dynamic. For example, an online evaluator like ours folds

```
if x == 23 { T } else { F }
```

to `T` if `x == 23`. A BTA only knows that `x` is static but not what value. Furthermore, when dealing with higher-order programs as we do, a BTA requires a CFA (see Section 10.6) which might introduce even more imprecisenesses: as argued in Section 10.6, in our setting an *on-the-fly* CFA is more precise than a CFA that runs *once beforehand* because evaluating the program yields full context-sensitivity. In addition, a CFA can be quite costly depending on the desired precision.

Partial evaluation has also been used to optimistically optimize dynamic languages like Python, JavaScript [Wür+13; Wür14; Hum+14; Wür+17; Wim+17], or R [Sta+16; Fum+17]. The implementation of the language consists of an interpreter that is aware of the partial evaluator. This means that the programmer must annotate the interpreter in several ways. For example: fields that are candidates for specialization, functions which should not be entered by the evaluator, or the AST data structure such that the evaluator understands which fields are the children of an AST node. After starting the interpreter for a given program, the runtime system might detect after running several iterations of a "hot" loop that the iteration variable was always an `int`. Assumptions like these will be specialized by partial evaluation. The resulting code is then just-in-time (JIT)-compiled to machine code in order to speed up the execution of such "hot code regions". Runtime checks are inserted to verify that the assumptions made still hold as the compiled version runs. If one of these assumptions is no longer correct, the code must be *deoptimized*. This means that control is transferred back to the interpreter.

10.2.2. Metaprogramming

Metaprogramming allows the programmer to write a program that generates another program. In other words, the programmer manually implements the generating extension P_{gen}. For this reason, metaprograms conceptually look like the pseudocode in Listing 10.3c and the programmer can explicitly stage a DSL interpreter [Cza+03]. Many projects implement P_{gen} in a scripting language like Python. The script is invoked at build-time

to generate P_{spec}—usually in a low-level language like C [e.g. Pro+13; Rat+17]. As such scripts simply splice strings, the residuum may be ill-typed. Another possibility is to run a C++ program as generator in order to construct a GPU program [HG14; Hai+16]. The focus of this line of work is to increase programmer productivity by only using C++ but also target GPU code generation. Other approaches like C++ template metaprogramming [Vel98], Terra [DeV+13] (see Listing 10.4b), (quasi-)quotation and macros in Scheme/Lisp, or Racket [Tob+11] increase programmer productivity by incorporating metaprogramming facilities into the language but still may construct an ill-typed residuum. MetaML [TS00] and MetaOCaml[2] (see Listing 10.4a) on the other hand, guarantee well-typedness of the residuum if the metaprogram is well-typed.

With partial evaluation, well-typedness of the residuum comes for free because type checking is independently performed prior to specialization. Moreover, the intrusive staging annotations make it difficult for the programmer to read and understand the program: The shape of the residuum is concealed by the metaprogram. To really understand what the residuum actually does, the programmer has to execute the metaprogram in her mind. With partial evaluation on the other hand, the programmer can ignore the different stages if she just wants to understand the program. Finally, since the stage is a feature of the syntax, functions are not *polyvariant*. This means that it is not possible to write a function that is polymorphic in the binding time of its parameters. Dynamic staging [Dan+14] tackles this problem by introducing the stage as a first-class citizen to the language at the cost of an unsound type system.

10.2.3. DSL Embedding

Hudak [Hud98] was one of the first to *embed* a DSL into a host language in order to inherit much of its infrastructure. Carette, Kiselyov, and Shan [CKS07] suggested to embed a typed language by ordinary functions instead of object terms. Based on this work, Hofer et al. [Hof+08] have described a *polymorphic* embedding that supports multiple interpretations including an optimizer as yet another interpretation. Rompf and Odersky [RO10] extend this work even further to lightweight modular staging (LMS) with a focus on performance-oriented DSLs [Cha+10] like OptiML [Suj+11] or

[2] see `http://www.cs.rice.edu/~taha/MetaOCaml/`

Liszt [DeV+11]. LMS does not rely on explicit staging capabilities of the *host* language Scala. Instead, executing the host program constructs a second domain-specific program representation like Delite [Bro+11] (*deep* embedding—see Section 8.1). Values of type `T` are wrapped into a type operator `Rep[T]` to represent values that should appear in the residuum. As the stage is encoded in the type system, Scala's type inference works akin to a local BTA [PS94]. Lancet [Rom+14] is an online partial evaluator for Java bytecode and serves as a front end for LMS. This is an alternative to explicit programming with `Rep` types. The deferred stage can also be executed at runtime in the setting of a JIT compiler. Comparable to other explicit metaprogramming techniques, LMS essentially requires the programmer to write the generating extension. However, via overloading and type inference the staged program is somewhere between P and P_{gen} (see Listing 10.4c). For example, `n % 2` is of type `Int`. Thus, the expression is executed when the host program runs. But since `r` is of type `Rep[Int]`, executing `r * r` results in a residuum containing a multiplication. The implementation of cogen lies in LMS' library, which implements `a * b` for `Rep[Int]`. The downside of this approach is that for data types unknown to LMS, the programmer must implement appropriate overloads ("cogen for these types") himself. To some extent, LMS can also be made polyvariant at the cost of introducing type variables for each desired staging combination [Ofe+13]:

```
def f[I[_]](i: I[Int])= { /*...*/ }
```

ArBB [New+11] and Halide [Rag+13] leverage a staging mechanism similar to LMS to construct the domain-specific program representation with C++ as host language.

HIPA^{cc} [Mem+16] and SYCL [Khr15], on the other hand, are *shallowly* embedded DSLs in C++. These DSLs rely on a modified C++ compiler that recognizes domain-specific patterns and manipulates the program representation in order to achieve performance. Jovanovic et al. [Jov+14] present a technique based on Scala macros to generate a deep embedding from a shallow one. An alternative is to use a partial evaluator that removes the overhead of the shallow embedding [Hud98; CKS07; Lei+15]. This approach is attractive as no dedicated DSL compiler is required. For this reason, this part of the thesis focusses on this idea. However, we have not found a partial evaluator, whose behavior is predictable for the programmer, that reliably supports higher-order functions, and can be easily integrated

```
fn count(i: int, n: int) -> int {
    if i < n {
        count(i+1, n)
    } else {
        i
    }
}
```

Listing 10.5.: The counting loop problem

into a performance-oriented programming language. This has motivated the work presented in this thesis.

10.2.4. Divergence in Partial Evaluation

Katz and Weise [KW92] distinguish three classes of divergence that may occur during partial evaluation:

True divergence: If the full evaluation of the program does not terminate for some inputs, partial evaluation might also not terminate.

Hidden divergence: A program may contain unreachable code that is divergent. Partially evaluating this divergent code may cause the partial evaluator to diverge.

Induced divergence: The partial evaluator diverges although neither true nor hidden divergences are present because the evaluator is too greedy. Consider the *counting loop problem* [BJ93] (see Listing 10.5): an aggressive partial evaluator might infinitely expand `count(0, d)` if `d` is dynamic, although full evaluation terminates for every $d \geq 0$.

It is easy to avoid all forms of divergence if recursive calls are not specialized. *Hybrid partial evaluation* [SC11] on the other hand, does not give any termination guarantees. For this reason, hybrid partial evaluation uses annotations similar to our approach.

A well-known technique [Fut99; CL11] we also employ to avoid at least obvious endless recursions, is to memoize each specialized call site. If the evaluator is about to specialize a memoized call, it will reuse that call instead of specializing again. However, this technique does not prevent the counting loop problem.

LMS has special behavior for a `while` loop: If the termination condition is of type `Boolean`, the loop will be running when the host program runs; if it is of type `Rep[Boolean]`, LMS will construct a residual loop. This approach only works since Scala does not support *unstructured control flow* as we do. In particular, the programmer cannot use `break` or `continue` within a loop.[3] LMS also leverages the aforementioned memoization technique for recursive calls. This has the effect that a counting `while`-loop with a dynamic conditional terminates when the host program runs, whereas a recursive implementation with an `Int` counter and a `Rep[Int]` bound diverges.

Both Similix [Jør98] and Schism [Con93] are offline evaluators using BTA. These evaluators will not evaluate a cycle if the condition that breaks the cycle remains dynamic. On the one hand, this is slightly more aggressive than our approach because our approach will also jump over an acyclic, dynamic conditional. On the other hand, both evaluators suffer from the inherent imprecisions caused by the BTA and CFA (see above).

Other more complex termination heuristics, like monitoring the argument sizes of recursive calls, have been applied in the past [JG02; JGS93]. We believe such heuristics are difficult for the programmer to understand.

10.3. Full and Partial Evaluation

The relation $\Rightarrow$ (see Figure 10.1 and Appendix A for the notation) defines a call-by-value semantics. It is similar to $\rightarrow$ (see Figure 9.6) but deterministic. E-Eval corresponds to the congruence rules (R-Cong* and R-Primop) but evaluates all subterms in a deterministic order until they are in normal form. E-Where, E-App, and E-Fold correspond to R-Where and R-App, respectively, but require all subterms to be in normal form in order to fire. E-Skip is the rule where partial evaluation kicks in. If a call in normal form cannot be folded via E-App because the callee e_0 is not a closure, evaluation proceeds at the post-dominator (see Section 10.3.2). These insights lead to the following lemma:

[3] There are workarounds in Scala, which leverage exceptions, but these workarounds do not work well with LMS.

Evaluation context:

$$\mathcal{E}[\star] \coloneqq \star \texttt{ where } \overline{f} \mid \star(\overline{e}) \mid e_0\Downarrow\ (e_1\Downarrow, \dots, \star, \dots, e_n) \mid \boxtimes(e_1\Downarrow, \dots, \star, \dots, e_n)$$

$$\text{E-Eval}\ \frac{a \Rightarrow a'}{\mathcal{E}[a] \Rightarrow \mathcal{E}[a']}$$

$\boxed{b \Rightarrow b'}$

$$\text{E-Where}\ \frac{b_0\Downarrow \quad \cdots \quad b_n\Downarrow}{\overbrace{b_0 \texttt{ where } \ell_1(\overline{x^1 : t^1}) : b_1, \dots, \ell_n(\overline{x^n : t^n}) : b_n}^{\overline{f}} \to [\ell_1 \mapsto \ell_1(\overline{x^1 : t^1}) : b_1 \texttt{ where } \overline{f}] \cdots [\ell_n \mapsto \ell_n(\overline{x^n : t^n}) : b_n \texttt{ where } \overline{f}]b_0}$$

$$\text{E-App}\ \frac{e_1\Downarrow \quad \cdots \quad e_n\Downarrow}{(\ell(\overline{x : t}) : b)(e_1, \dots, e_n) \Rightarrow [x_1 \mapsto e_1] \cdots [x_n \mapsto e_n]b}$$

$$\text{E-Skip}\ \frac{e_0\Downarrow \quad \cdots \quad e_n\Downarrow \qquad e_0 \text{ not a closure} \qquad \ell_p(\overline{x : t}) : b_p = \mathit{postdom}(e_0(e_1, \dots, e_n))}{e_0(e_1, \dots, e_n) \Rightarrow b_p}$$

$\boxed{e \Rightarrow e'}$

$$\text{E-Fold}\ \frac{n > 0}{\boxtimes(\nu^1_{t1}, \dots, \nu^n_{tn}) \Rightarrow \otimes(\nu^1_{t1}, \dots, \nu^n_{tn})}$$

Figure 10.1.: Evaluation ($\Rightarrow$) in λ^{cps}

Lemma 10.1 ($\Rightarrow \subset \to$)

$$\textit{For all}\ \begin{matrix} b, b' \\ e, e' \end{matrix}\ \textit{if}\ \begin{matrix} b \Rightarrow b' \\ e \Rightarrow e' \end{matrix}\ ,\ \textit{then}\ \begin{matrix} b \to b' \\ e \to e' \end{matrix}\ .$$

Proof. By a straightforward mutual induction on a derivation of $\begin{matrix} b \Rightarrow b' \\ e \Rightarrow e' \end{matrix}$. □

Lemma 10.2 (λ^{cps}: $\Rightarrow$-Progress)
Every λ^{cps} term is either a term value or can be stepped with $\Rightarrow$ into another term. To be more precise:

$$\textit{If}\ \begin{matrix} \Gamma \vdash b \\ \Gamma \vdash e : t \end{matrix}\ ,\ \textit{then}\ \begin{matrix} b = \texttt{exit}(\nu) \\ e = \nu \end{matrix}\ \textit{or}\ \begin{matrix} \exists b' : b \Rightarrow b' \\ \exists e' : e \Rightarrow e' \end{matrix}\ .$$

Proof sketch. By mutual induction on a derivation of $\begin{array}{c}\Gamma \vdash b \\ \Gamma \vdash e : t\end{array}$.
See Section B.3 for the full proof.

Lemma 10.3 (λ^{cps}: $\Rightarrow$-Preservation)
As we reduce a λ^{cps} term with $\Rightarrow$, its type is preserved at each step. To be more precise:

$$\textit{If} \quad \begin{array}{c}\Gamma \vdash b \\ \Gamma \vdash e : t\end{array} \quad \textit{and} \quad \begin{array}{c} b \Rightarrow b' \\ e \Rightarrow e'\end{array} \quad \textit{then} \quad \begin{array}{c}\Gamma \vdash b' \\ \Gamma \vdash e' : t\end{array} \quad .$$

Proof. By Lemma 9.2 and Lemma 10.1. □

Theorem 10.1 (λ^{cps}: $\Rightarrow$-Soundness)
No well-typed λ^{cps} term gets stuck while reducing with $\Rightarrow$. To be more precise:

$$\textit{If} \quad \begin{array}{c}\Gamma \vdash b \\ \Gamma \vdash e : t\end{array} \quad \textit{and} \quad \begin{array}{c} b \Rightarrow^* b' \\ e \Rightarrow^* e'\end{array} \quad \textit{then} \quad \begin{array}{c} b' = \texttt{exit}(\nu) \\ e' = \nu\end{array} \quad \textit{or} \quad \begin{array}{c}\exists b'' : b' \Rightarrow b'' \\ \exists e'' : e' \Rightarrow e''\end{array} \quad .$$

Proof. By Lemma 10.2 and Lemma 10.3. □

10.3.1. Full Evaluation

If we evaluate a well-typed program, we will never have to handle the case that an expression does not reduce to a constant. In particular, when stumbling upon a call in normal form during evaluation, the callee e_0 will be a closure and consequently E-Skip will never trigger.

Definition 10.1 (Valid Configuration)
Let $\ell(\overline{x : t}) : b$ be a continuation without free variables. We call an argument sequence $\overline{\nu}$ of values a valid configuration for ℓ iff the application $\ell(\overline{\nu})$ is well-typed. The set $\mathcal{C}(\ell)$ denotes all valid configurations for ℓ.

Lemma 10.4 (Full Evaluation)
Let ℓ be a continuation value (see Definition 9.6). Evaluating $\ell(\overline{\nu})$ where $\overline{\nu} \in \mathcal{C}(\ell)$ never triggers E-Skip.

Proof. See proof of Lemma 10.3. □

10.3.2. Partial Evaluation

Reconsider Kleene's [Kle38] s_n^m-theorem (see Section 10.2): Partial evaluation means to reduce terms that may still contain free variables. This what we will discuss now.

Suppose a body b contains the free variable x and partial evaluation of b yields a new body b' that still contains x. Since x is bound by f in the programs

$$f(x : \texttt{int})\ b \qquad \text{and} \qquad f(x : \texttt{int})\ b'\ ,$$

both programs do not contain any free variables. If for each input ν_i, for which $f(\nu_i)$ reduces to $\texttt{exit}(\nu_r(\nu_i))$,

1. $f'(\nu_i)$ reduces to the same $\texttt{exit}(\nu_r(\nu_i))$, we know that partial evaluation is correct, and

2. b reduces in finitely many steps to a normal form b', we know that partial evaluation is neither subject to induced nor to hidden divergence (see Section 10.2.4).

The rest of this section discusses how $\Rightarrow$ also partially evaluates a program and proves that $\Rightarrow$ indeed satisfies both properties. But first we need to clarify how exactly we can get rid of free variables.

We have already discussed free variables in Section 9.3.2. We assume that the type of a free variable is known. The function $\overline{FV}(b)$ determines the free variables of b like FV but puts them additionally into an arbitrary sequence along with their associated types. *Lambda lifting* [Joh85] eliminates free variables by wrapping all free variables in a body into a continuation. Section 11.1 thoroughly discusses lambda lifting.

Definition 10.2 (Lambda Lifting)
Let $\overline{FV}(b) = \overline{x : t}$ be the sequence of free variables in a body b well-typed under $\overline{FV}(b)$. Then, $\lambda b \coloneqq \ell(\overline{x : t}) : b$ is the lambda-lifted continuation of b.

Let us come back to rule E-Skip. This rule uses a function $\mathit{postdom}(b)$ to get a post-dominator of b if the callee e_0 is not a closure. Evaluation will continue at that post-dominator. For the time being, we are not interested in *how* post-dominators are computed. We will discuss this in Section 10.6. We merely require $\mathit{postdom}(b)$ to exist and to compute a *valid* but not necessarily the *immediate* post-dominator of b.

Definition 10.3 (Post-Dominator)
Let b be a body well-typed under $\overline{FV}(b)$. We call ℓ_p a *post-dominator* of b iff $\ell_p = \top$ or all possible finite evaluations from b to any exit visit ℓ_p:

$$\forall \overline{\nu} \in \mathcal{C}(\lambda b): \quad (\lambda b)(\overline{\nu}) \Rightarrow^* \texttt{exit}\ e\!\Downarrow \quad \text{implies} \quad (\lambda b)(\overline{\nu}) \Rightarrow^* \ell_p(\cdots) : b_p \ .$$

Remark. Hence, $\top$ designates the point in execution after all $\mathit{exits}(b)$ and is always a valid post-dominator—even for diverging programs. This is a common trick in compilers to implement a sane post-dominance analysis for programs with no or multiple exits. See Section 10.6.4 for a thorough discussion.

If we lambda-lift all free variables into a continuation λb, all variables in λb will be bound and we can perform a full evaluation (see Lemma 10.4). If we perform n steps on b, we will partially evaluate b and obtain b'—possibly by using E-Skip. If we lambda-lift b', we can perform a full evaluation on that program. The following theorem states that both bodies still compute the same result.

Theorem 10.2 (Correctness)
Let b be a body well-typed under $\overline{FV}(b) = \overline{x : t}$. Furthermore, let $b \Rightarrow^n b'$ and ℓ defined as $\ell(\overline{x : t}) : b'$. For each input $\overline{\nu}$, for which $(\lambda b)(\overline{c})$ terminates with result $\nu_r(\overline{\nu})$, $\ell(\overline{\nu})$ terminates with the same result.

$$\forall \overline{\nu} \in \mathcal{C}(\lambda b): \qquad (\lambda b)(\overline{\nu}) \Rightarrow^* \texttt{exit}\ \nu_r(\overline{\nu}) \quad \textit{implies} \quad \ell(\overline{\nu}) \Rightarrow^* \texttt{exit}\ \nu_r(\overline{\nu}) \ .$$

Remark. Note that ℓ uses the free variables of b and not b'. This is because partial evaluation might eliminate some free variables, but we would like both λb and ℓ to have the same signature.

Proof. By Lemma 10.1 and Theorem 9.1. □

Remark. In fact, any partial evaluation strategy $R \subset \Rightarrow$ is correct!

Finally, the following theorem states that partial evaluation is neither subject to induced nor to hidden divergence:

```
@f(args);
/*next*/;
```

(a) Impala

```
@f(args, k) where
  k(/*...*/): b_next
```

(b) λ^{cps}

Listing 10.6.: Run-annotated call

Theorem 10.3 (Termination Guarantee)
Let b be a body well-typed under $\overline{FV}(b)$. For each input $\overline{\nu}$, for which $(\lambda b)(\overline{c})$ terminates, partially evaluating b terminates to a normal form b' in finitely many steps:

$$\forall \overline{\nu} \in \mathcal{C}(\lambda b): \qquad (\lambda b)(\overline{\nu}) \Rightarrow^* \texttt{exit}\ \nu_r(\overline{\nu}) \quad \textit{implies} \quad b \Rightarrow^* b' \not\Downarrow \ .$$

Proof. By Lemma 10.4 and Definition 10.3.

□

10.4. Run and Halt Annotations

Until now, we have studied partial evaluation of a whole program. But in practice, the programmer only wants to specialize certain parts of the program while other parts should be excluded from specialization (see Section 10.7). For this reason, we introduce a *run* annotation that causes

```
@f(args, k)
```

to be specialized by triggering rule E-App (see Listing 10.6). As partial evaluation should only run until a continuation (e.g. `k`) outside the body is called, all remaining free variables, together with any **`exit`**, must be considered local *exits* for the purpose of evaluation and the definition of post-dominators in particular (see Section 10.6). Similarly, a *halt* annotation

```
$g(args, l)
```

causes the evaluator to stop specialization at that point and resume evaluation at `l`.

10.4.1. Termination Implications

Run annotations impact the termination of partial evaluation. If a *run*-annotated code block is unreachable, the partial evaluator might be subject to hidden divergence:

```
fn count(i: int, N: int)
          -> int {
  if i < N {
    @count(i+1, N)
  } else {
    i
  }
}
```

(a) Impala

```
count(i: int, N: int,
      ret: cn(int)):
  br(i<N, T, F) where
    T():
      @count(i+1, N, ret)
    F():
      ret(i)
```

(b) λ^{cps}

```
count(i: int, N: int, ret: cn(int)):
  br(i<N, T, F) where
    T():
      br(i+1<N, T', F') where
        T'():
          @count(i+2, N, ret)
        F'():
          ret(i)
    F():
      ret(i)
```

(c) After one partial evaluation run

Listing 10.7.: Run-annotated recursive call

```
if i_am_always_false_but_the_compiler_does_not_know() {
    @i_will_not_terminate(42)
}
```

However, trivial unreachable code will be removed by the compiler, so the following annotated call will be removed before the partial evaluator runs:

```
if false {
    @i_will_not_terminate(42)
}
```

Under certain circumstances, run annotations might induce divergence. Reconsider the function `count` from Listing 10.5 where the recursive call is annotated with `@` (see Listing 10.7a–b). Substitution of `count` and application of E-App renders Listing 10.7c. This program contains a new call `@count`. After partial evaluation has terminated, the compiler looks for the next run annotation and stumbles upon that `@count`. Consequently, rule E-App triggers again, causing the compiler to diverge. This divergence does not violate Theorem 10.3. As already stressed, the theorem merely guarantees that *one* partial evaluation run will not induce divergence.

This situation only occurred because a *recursive call* had been annotated. A call to a recursive continuation outside its definition is *not* recursive and, thus, does *not* induce divergence. Hence, annotating all other calls to `count`, like `@count(0, N, RET)`, is not problematic. We would like to stress that a single partial evaluation run is already Turing-complete. Therefore, the partial evaluator completely folds `@ackermann(3, 4)` to `125`. As outlined above, this call is *not* recursive.

If we disallow recursive bindings in the **`where`**-clause of λ^{cps}, the resulting language will not have a fixed-point operator anymore. Thus, the resulting language is just like the simply-typed lambda calculus *strongly normalizing* [Tai67]. In particular, it is no longer possible to replicate a fixed-point operator like the Y combinator as in the untyped lambda calculus. In reverse, this means that the compiler can statically over-approximate *all* recursions in a λ^{cps} program. Thus, the compiler can warn the programmer about potentially dangerous run annotations.

```
fn foo(mut a: [float * 2],
       i: int, cond: bool)
       -> int {
    if cond {
        a(0) = 42
    } else {
        ++a(1)
    }
    a(i)
}
```

(a) Impala

```
foo(a: [int * 2],
    i: int, cond: bool,
    ret: cn(int)):
  br(cond, T, F) where
    T():
      N(ins(a, 0, 42))
    F():
      N(ins(a, 1, ex(a, 1)+1))
    N(b: [int * 2]):
      ret(ex(b, i))
```

(b) λ^{cps}

Listing 10.8.: Functional arrays

10.5. Dealing with Compound Data Types

Consider the imperative program in Listing 10.8a, which resorts to the mutable array `a`. As long as the address of a variable is not taken, the Impala compiler translates a mutable variable with state into a stateless functional program (see Listing 10.8b). Each lookup translates to an `extract` operation in λ^{cps}, which conceptually creates a copy of the indexed element. Each update translates to an `insert` operation, which conceptually creates a copy of the whole array where the element in question has been updated. When different values from different control-flow paths meet, the Impala compiler installs a parameter to that continuation (see `N`'s parameter `b`). This is akin to a Φ-function in SSA form (see Section 9.1.4). Indeed, the Impala compiler uses an SSA construction algorithm for this translation [Bra+13].

Now suppose the partial evaluator would like to specialize

```
foo([x, 22], 1, false, k)
```

where `x` is some value unknown to the evaluator and `k` is some known continuation. As `cond` is `false` the evaluator continues in `F` and computes:

```
N(ins([x, 22], 1, ex([x, 22], 1)+1))
N(ins([x, 22], 1, 22+1))
N(ins([x, 22], 1, 23))
N([x, 23])
```

This in turn causes the evaluator to enter `N`:

```
k(ex([x, 23], 1))
k(23)
```

Note how the whole array has collapsed and the evaluator will continue with `k(23)`. In this fashion, Impala supports arrays, `structs`, and tuples, which may be nested in any combination.

10.6. Local Control-Flow Analysis

So far, we have assumed that the *postdom* continuation in rule E-Skip will obey Definition 10.3. The prerequisite for computing post-dominance is a CFG. If the input program is of first order, continuations cannot be passed as arguments to other continuations. Hence, all callees are statically known and it is straightforward to construct a CFG from that program.

But higher-order programs may call parameters. In order to compute a CFG in this setting, we need to statically know, which continuations may actually reach a parameter at runtime. A k-CFA [Shi91; NN92; Mid12] computes this information for calling contexts of length k. With increasing k the analysis becomes more costly but also more precise. Hence, a 0-CFA is context-insensitive, leading to imprecisions as continuation arguments are merged from all calling contexts.

One possibility would be to apply a k-CFA prior to partial evaluation in order to determine an appropriate post-dominator in the context of a dynamic branch or call. However, for calls deeper than k contexts, the analysis information becomes imprecise. The computed post-dominator might then lie closer to the *exits*, causing the evaluator to skip more code than necessary. The programmer would have a hard time to understand, track, and work around such imprecisions.

For this reason, our partial evaluator follows a different approach. As long as the evaluator does not hit a dynamic branch or call (see E-Skip), the evaluator does not need any post-dominance information. Until such a point, evaluation will have expanded all calls, resulting in full context-sensitivity. Whenever the evaluator needs to apply rule E-Skip, it runs a CFA. This enables construction of a CFG to compute a post-dominator. The CFA we employ is *local*. This means that it starts at the run-annotation and only analyzes continuations currently declared within that scope. We call these continuations *inside* continuations. We call continuations declared outside the scope, in particular continuations defined in other translation units or intrinsic higher-order continuations such as `br` or `nvvm` (see Section 10.7.1), *outside* continuations. Suppose partial evaluation of program `start` to `end`

in Listing 10.9 meets a dynamic branch in `f`. The continuation `out` and the higher-order parameter `end` are not defined within `start`'s scope and are thus outside continuations. All other continuations are inside.

To begin with, we describe a simple CFA, explain the reason for its impreciseness, and then motivate a better CFA.

10.6.1. Non-Symbolic CFA

A simple (*non-symbolic*) CFA works as follows: The CFA handles inside continuations context-insensitively. References to outside continuations and free variables are unknown and yield `⊤`. So propagation from `F` through `C` yields the value `N` for parameter `pb`, which is precise. However, this CFA must propagate `⊤` to the parameters of the higher-order continuations `A` and `C`. This results in the CFG shown in Figure 10.2a, which in turn yields the post-dominator `⊤`—the unique local exit node in the CFG (see Definition 10.3). Partial evaluation would then skip the remainder of the scope.

10.6.2. Symbolic CFA

To precisely deal with references to outside continuations, the CFA deals with them *symbolically*: each outside continuation's control flow is represented via one symbolic value (see Figure 10.2b). Have a look at the call to `out`, which receives `A` and `C` as arguments. As the CFA does not analyze `out`, the CFA must assume that continuations within `out`'s scope might call `A` and `C`. For this reason, the CFA draws the edges `out` → `A` and `out` → `C`. Furthermore, the CFA propagates `out` as value for `A`'s parameter `pa` and `C`'s parameter `pc`. As `A` calls `pa` and `C` calls `pc`, the CFA draws the edges `A` → `out` and `C` → `out`.

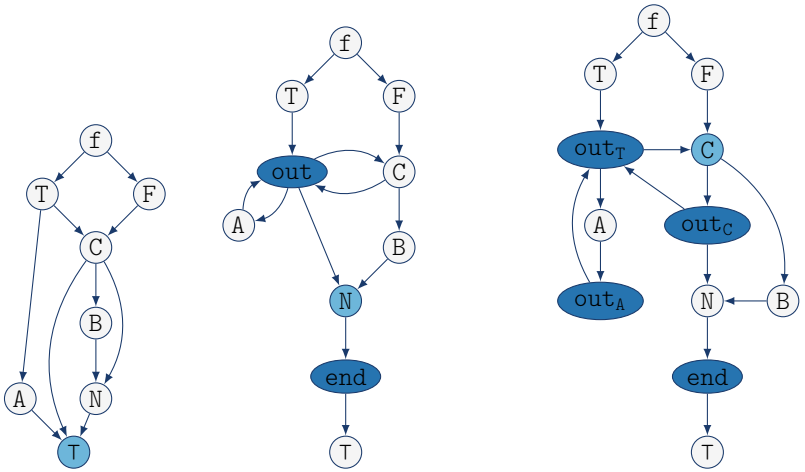
Note that this CFA merges all control flow over all contexts for `out` into one single CFG node. This yields the imprecise post-dominator `N`.

10.6.3. Partially Context-Sensitive CFA

Instead of merging the arguments over all calls when constructing the CFG, we give one level of context to outside continuations. Thus, the resulting CFA creates a node $\texttt{out}_\texttt{T}$ for the call to `out` within `T` and propagates $\texttt{out}_\texttt{T}$ as value for `pa` and `pc`. This value ends up in the call in `A`. This instantiates

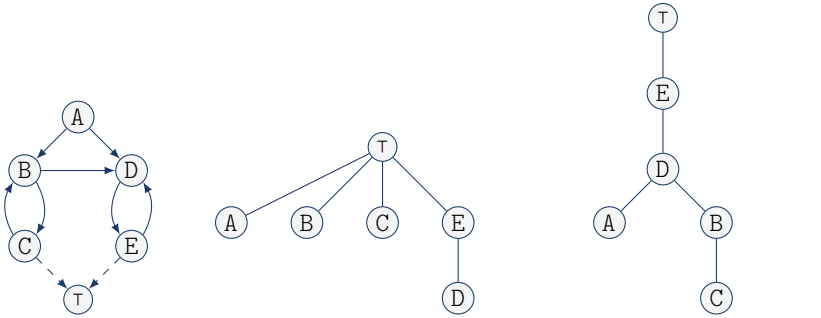
```
start(/*...*/, end: cn()):
  /* program already evaluated */
  f():
    br(/*dynamic*/, T, F) where
      F():
        C(B)
      T():
        out(A, C) where
          A(pa: cn(int)):
            pa(42)
          B(pb: cn()):
            pb()
          C(pc: cn(cn())):
            pc(N) where
              N():
                end()
```

Listing 10.9.: Partial evaluation from `start` to end hits a dynamic branch in `f`.



(a) Non-symbolic (b) Context-insensitive (c) Partially context-sensitive

Figure 10.2.: CFGs obtained by applying various flavors of CFAs on Listing 10.9. Light nodes represent inside, dark nodes represent outside continuations. The marked inside node is the resulting post-dominator of `f` in its respective graph.



(a) CFG (b) Postdom-tree *with* $C \to \top$ (c) Postdom-tree *without* $C \to \top$

Figure 10.3.: Fake edges (dashed) in a CFG not connected to the exit ($\top$). The fake edge $C \to \top$ is superfluous.

$\texttt{out}_\texttt{A}$ and causes the edges $\texttt{out}_\texttt{A} \to \texttt{out}_\top$ and $\texttt{A} \to \texttt{out}_\texttt{A}$. Similarly, the CFA draws the edges $\texttt{out}_\texttt{C} \to \texttt{out}_\top$ and $\texttt{C} \to \texttt{out}_\texttt{C}$. Hence, the CFG in Figure 10.2c separates the calls to `out` in `T`, `A` and `C`. From this underlying graph, `C` results as post-dominator of `f`.

In these cases, however, the partial evaluator cannot guarantee termination anyway (see Theorem 10.3). Continuing evaluation from the computed post-dominator remains sound for returning cases.

10.6.4. Finding a Unique Exit

In practice a CFG obtained by a CFA does not necessarily contain a unique *exit*. The CFG may contain multiple exits or no exit at all. However, for a post-dominance analysis, we need a *unique* exit and each node must be reachable from that exit in the *reverse* CFG that is retrieved by reversing the direction of all edges.

For a CFG with multiple exits we simply introduce a virtual exit node $\top$ and draw fake edges from each exit to $\top$. If a CFG (or one of its subgraphs) does not contain exits at all, we need a different strategy to connect the graph to $\top$. The CFG in Figure 10.3a, for example, is not connected to $\top$. A naïve approach is a post-order walk of the CFG and to connect each node that is not reachable from $\top$ in the reverse CFG. A valid post-order walk is $CEDBA$. Since both C and E are not reachable from $\top$ in the

reverse CFG, we draw the fake edges $C \to \top$ and $E \to \top$. Consequently, the post-dominator tree in Figure 10.3b emerges. Another valid post-order walk is $EDCBA$. By first adding the edge $E \to \top$, C becomes reachable and we do not draw the edge $C \to \top$. For this reason, we can compute a more precise post-dominator tree (see Figure 10.3c).

The algorithm in Listing 10.10 performs a post-order walk. Each time it becomes evident that a fake edge is necessary, the algorithm tries to backtrack in order to find a different post-order walk. By doing so, this algorithm never draws superfluous fake edges such as $C \to \top$ in Figure 10.3a.

10.7. Evaluation

We demonstrate the effectiveness of our partial evaluator on two shallowly embedded DSLs:

1. We present a small framework for stencil computations in image processing: The framework is essentially an "interpreter" that applies a stencil to an image. The aspects of boundary handling, application of the stencil, and the stencil itself are cleanly separated. Partial evaluation composes those aspects together and produces high-performance code that we specialize for execution on CPU and GPU targets.

2. We present a DSL for the V-cycle multigrid iteration; a multigrid method important in high-performance computing. The V-cycle employs different stencils to smooth the error on different resolutions of the same data. Passing the V-cycle components as functions to the DSL allows us to merge multiple components in order to reduce high latency memory accesses.

These DSL embeddings have been developed in collaboration with Richard Membarth [Lei+15]. He also provided the HIPA^{cc} implementations for the V-cycle. The focus on our side was the work on the compiler.

Before delving into the DSLs, we will discuss how a machine expert (see Section 8.2) employs Impala in order to abstract from hardware-dependent details of the target hardware.

```
function connect_to_exit
    stack ← empty stack

    function backwards-reachable(n)
        mark all nodes reachalbe from n in reverse(G) as
         reachable-from-exit
    end

    function backtrack(n)
        backtrack-stack ← empty stack
        candidate ← nil
        for n ∈ post-order(reverse(G), ⊤, backtrack-stack) :
            if n ∈ stack and n not done during outer postorder-walk
                candidate ← n
            end
        if candidate
            i ← find(stack.begin(), stack.end(), candidate)
            move(i + 1, stack.end(), i)
            stack.back() ← candidate
            return true
        end
        return false
    end

    backwards-reachable(⊤)
    for n ∈ post-order(G, entry, stack) :
        if n not reachable-from-exit
            if not backtrack(n)
                draw n → ⊤
                backwards-reachable(n)
            end
        end
end
```

Listing 10.10.: Backtracking algorithm to draw fake edges to exit

```
fn iterate(field: Field, body: fn(int, int) -> ()) -> () {
    let vector_length = 8;
    for y in range(0, field.rows) {
        for x in vectorize(vector_length, 0, field.cols) {
            body(x, y);
        }
    }
}
```

Listing 10.11.: Iterator implementation for SIMD hardware

10.7.1. Dealing with Hardware Details

Mapping Algorithms to Different Architectures

In order to abstract from specific target platforms, Impala provides intrinsic higher-order functions. For example, invoking the following function enters SIMD mode of vector length `L` (see Part I) for `body` and creates an appropriate loop from `a` to `b`:

```
fn vectorize(L: int, a: int, b: int, body: fn(int) -> ()) -> ()
```

Internally, Thorin instruments RV for the vectorization (see Section 2.3). Likewise, invoking the following function causes `body` to be executed via NVVM [NVI14] on an NVIDIA GPU:

```
fn nvvm(grid:  (int, int, int),
        block: (int, int, int),
        body: fn() -> ()) -> ()
```

The execution runs in parallel by the threads defined by `grid` with the given `blocking`. Similarly, Impala supports code generation for CUDA [NVI17], OpenCL [Khr12], and SPIR [Khr14]. In contrast to pragma-based solutions like OpenACC or OpenMP, Impala's intrinsics integrate seamlessly with Impala's type system. This allows the programmer to hide the use of these functions behind other functions. Reconsider the higher-order function `iterate` from Listing 8.4b in order to iterate over a field:

```
iterate(fld, |x, y| { /* some loop body */ });
```

Listing 8.4c depicts an implementation that schedules the loop body on the GPU. Listing 10.11 is a different implementation that vectorizes the loop

body. Other iterator implementations may use other intrinsics and/or more sophisticated blocking schemes.

Memory Management

Example 10.2 (Free Variables in Accelerator Intrinsic)
A function passed to an intrinsic like `nvvm` typically contains free variables:

```
let mut array: &[float] = /*...*/;
with nvvm(/*...*/) {
    array(i) = /*...*/;
}
```

Impala currently supports scalar variables, arrays, tuples, and `structs` to be used across devices, as long as they do not contain pointers. All required memory transfers from the host device to the accelerator and back are automatically generated by Impala. More fine-grained control is possible by the provided `mmap` and `munmap` functions.

Example 10.3 (Memory Management)
The following code transfers memory to the *global* device memory on the first GPU and releases it afterwards:

```
let mut array: &[float] = /*...*/;
let gpu_array = mmap(array, GPU0, Global, offset, size);
with nvvm(/*...*/) {
    gpu_array(i) = /*...*/;
}
munmap(gpu_array);
```

This allows fine-grained control over memory lifetimes and transfers between host and device memory. In a similar fashion, different memory types like read-only *texture* memory or on-chip *shared* memory can be utilized. These memory types are tuned for specific use-cases and can speed up a program significantly.

	GTX 970				Radeon R9 290X		Iris 5100		Core i5-4288U	
	NVVM		OpenCL		SPIR		OpenCL		CPU	AVX
LU	–	✓	–	✓	–	✓	–	✓	–	–
SS	2.34	2.26	2.34	2.26	1.02	0.97	17.49	17.15	85.69	155.57
+ BH	2.36	2.30	2.38	2.28	1.05	0.99	17.00	16.87	23.56	23.23
+ SM	1.61	**1.28**	1.67	**1.27**	0.82	**0.76**	24.21	**12.84**	16.67	**15.98**
OCV 2.4	2.24		2.17		0.89		18.55		27.21	
OCV 3.0	2.24		2.11		1.42		16.61		26.63	

^{LU} loop unrolling; "–": disabled, "✓": enabled
^{SS} stencil specialization
^{BU} boundary handling
^{SM} scratchpad memory
^{OCV} OpenCV version used

Figure 10.4.: Execution times in ms for the Gaussian blur of size 5×5 on an image of 4096×4096 pixels

10.7.2. Stencil Computations

A linear filter convolves an image with a filter mask by applying the filter mask to each pixel. Examples of linear filters are the Gaussian blur filter, the Laplace operator, or the Sobel operator. Since the filter mask for linear filters like the Gaussian blur or the Sobel operator are separable, we split a filter mask of $N \times M$ in a row and column component of size $N \times 1$ and $1 \times M$, respectively. This reduces the number of required memory accesses from $N \cdot M + 1$ to $N + M + 2$ and is, hence, used for our implementation.

Stencil Specialization (SS)

We describe linear filters using the `apply_stencil` function of our stencil framework. This function receives a filter mask and applies it to a field. Using the *run*, annotation we create a specialization for a given linear filter where the filter values are propagated into the code instead of reading them from memory:

```
let stencil: Stencil = /* Gaussian blur *:
let mut out: Field   = /* ... */;
for x, y in iterate(out) {
    out(y)(x) = @apply_stencil(x, y, arr, stencil, /*...*/);
}
```

```
fn iterate(out: Field, body: fn(int, int) -> ()) -> () {
    let unroll_factor = 4;
    let grid  = (out.cols, out.rows/unroll_factor, 1);
    let block = (128, 1, 1);
    with nvvm(grid, block) {
        let x = tid_x() + ntid_x()*ctaid_x();
        let y = tid_y() + ntid_y()*ctaid_y()*unroll_factor;
        for i in @range(0, unroll_factor) {
            body(x, y + i * ntid_y());
        }
    }
}
```

Listing 10.12.: Loop unrolling

All variants in Figure 10.4 are specialized in this way for the Gaussian blur.

Loop Unrolling (LU)

The discussed `iterate` functions abstract the iteration order. On a GPU, for example, it is beneficial to process multiple pixels by the same thread. To achieve this, we call `body` that is passed to `iterate` multiple times for different iteration points. This unrolls the iteration space by `unroll_factor` (`4` in the example below). We keep the implementation parametric in its unroll factor and use partial evaluation for the actual unrolling (see Listing 10.12). Variants in Figure 10.4 checked in column *LU*, use this technique.

Boundary Handling (BH)

To handle array boundaries, we clamp the index to the last valid element within the array in the `apply_stencil` function. Considering the row component, we need only to apply boundary handling at the left and right border of the image. Therefore, we introduce a `region` parameter to `apply_stencil` (see Listing 10.13a). Now, the `iterate` function iterates over different regions of each line instead of naïvely iterating over the whole image. Due to the *run* annotation on the loop, which iterates over the regions, boundary checks will only appear in the residual code for the left and right regions and will be specialized into `apply_stencil`. The programmer passes the functions for boundary handling as higher-order arguments to the

```
type boundary_handler = fn(int, int, int, cn(float)) -> int;

fn apply_stencil(region: int, /*...*/,
                 bh_lower: boundary_handler,
                 bh_upper: boundary_handler) -> float {
    // ...
    if region==0 { // left
        x = bh_lower(x, 0, arr.cols, return);
    }
    if region==2 { // right
        x = bh_upper(x, 0, arr.cols, return);
    }
    // ...
}

fn iterate(/*...*/) -> () {
    let limits = /* lower and upper limits for each region */;
    for y in $range(0, out.rows) {
        for region in @range(0, 3) { // left, center, right
            let bounds = limits(region);
            for x in $range(bounds(0), bounds(1)) {
                @body(x, y, region);
            }
        }
    }
}
```

(a) apply_stencil

```
fn clamp_lower(idx: int, low: int, up: int,
               out: cn(float)) -> int {
    if idx < low { low } else { idx }
}

fn const_lower(idx: int, low: int, up: int,
               out: cn(float)) -> int {
    if idx < low { out(1.0f) } else { idx }
}
```

(b) Boundary Handler

Listing 10.13.: Boundary Handling

stencil DSL, for example `clamp_lower` or `const_lower` (see Listing 10.13b). The function `const_lower` always skips further computations in the case that `idx` lies outside the given range. `const_lower` achieves this by passing the constant `1.0f` to the continuation `out`. This is the **`return`** continuation in `apply_stencil`. During partial evaluation it is important to infer that the function passed to `out` is the proper post-dominator (see Section 10.6).

Scratchpad Memory (SM)

The stencil operation of the filter has high spatial locality and neighboring elements are read by multiple threads. Therefore, we can first load data for a group of threads to fast scratchpad memory (shared or local memory) and then read the neighboring elements from this scratchpad. This also allows us to fuse the row and column component of the filter into a single kernel. We use the scratchpad memory as output memory for the first component and as input memory for the second component. Fusing multiple components is outlined in Section 10.7.3.

Evaluation

For our measurements we use a separated version of the Gaussian blur filter with a 5×5 filter mask and an image of 4096×4096 pixels. All specialized versions are generated from the same generic description using partial evaluation. Figure 10.4 shows the median execution time in ms on the GTX 970 using the CUDA 7.5 drivers and toolkit, on the R9 290X using the Crimson 15.11 drivers as well as on the Iris 5100 and on the Intel Core i5-4288U on a MacBook Pro running OS X 10.11.2. On the discrete GPUs, the median of seven runs is used while 17 runs are used on the embedded GPU and 27 runs on the CPU.

The last two lines show the execution for hand-tuned CUDA, OpenCL, and CPU (vectorized C++) implementations from OpenCV (version 2.4.12 and 3.0.0), a state-of-the-art image processing toolbox. The CUDA implementation in OpenCV is provided by NVIDIA experts and uses similar optimizations: the filter is separated, the iteration space is unrolled, border handling is limited to thread blocks at the image border, and fast on-chip scratchpad memory is used to stage data. Different OpenCL implementations in OpenCV are provided by AMD and Intel experts: The former implementation merges the row and column components into a single kernel

whereas the latter keeps the row and column components as separate kernels. The merged kernel first loads the data to fast on-chip scratchpad memory and then executes the column component, storing its results again to the scratchpad memory. Afterwards, the row component is executed, loading its input from scratchpad and storing the result back to device memory. On the CPU, the row component is manually vectorized via double-pumped SSE instructions. The column component uses superword-level parallelism (SLP), unrolling multiple loop iterations such that the compiler can merge them easily into vector operations. The schedule always applies the row component first and then the column component. This allows the CPU to hold the intermediate results in cache.

It can be seen from Figure 10.4 that the specialized versions we obtain through partial evaluation even outperform the hand-tuned implementations in OpenCV. At the same time, our implementation is more concise: The hand-tuned CUDA version from OpenCV consists of 251 lines of CUDA code plus 330 kernel instantiations for different filter mask sizes and boundary handling modes. For OpenCL, OpenCV provides two different kernel implementations: one that merges the row and column component into a single kernel (142 lines of code) and one that implements the row and column component in separate kernels (278 lines of code). Boundary handling is realized via macros that wrap memory accesses. OpenCV's CPU implementations requires more than 1500 lines of code that consists of specialized implementations for different data types, kernel sizes, and target instruction sets (SSE, Neon). On the other hand, our implementation in Impala only requires 62 lines of code for the high-level algorithm and boundary handling description. The best performing CPU and GPU mappings only need 50 lines and 89 lines of Impala code, respectively.

We use the *run* annotations highlighted in the sample codes to trigger partial evaluation; none of them annotates a "dangerous" recursive call (see Section 10.4). We need *halt* annotations only to prevent loops that iterate over a field from being evaluated at compile time.

Note that we generate multiple NVVM/OpenCL kernels for the different boundary handling regions. These are executed serially by our runtime. Since the kernels at the image border process only a small amount of data, they do not utilize the GPU completely. Consequently, several of these kernels can also be executed in parallel, which would reduce our execution time further.

10.7.3. The V-Cycle Multigrid Solver

The basic idea of the multigrid method is to smooth the error (e. g., using an iterative method like Jacobi or Gauss-Seidel) on different resolutions of the same data. The *V-cycle* (see Listing 10.15) describes one possible multigrid iteration [BEM00; TS01]. To transform data between different resolutions of the multigrid, the algorithm leverages the *restrict* and *interpolate* methods. On each level, the error is smoothed (*smoother*) and estimated (*residual*). This process is recursive and starts at the finest resolution.

For a V-cycle DSL, we would like to have the different methods pluggable. Using Impala (see Listing 10.14), we pass the multigrid components as functions to `vcycle`. Furthermore, we use the `iterate` function introduced in Section 10.7.2. The partial evaluator inlines the multigrid components, unrolls the recursion and propagates other inputs (stencils, etc.). Furthermore, the evaluator weaves in the special higher-order functions for hardware-specific code generation. By providing specialized `iterate` implementations for CPU and GPU, we map the same algorithm to different target platforms.

This is a naïve implementation as each multigrid component is run after another. However, hand-tuned implementations of the V-cycle might merge multiple multigrid components in order to save unnecessary memory accesses. In Impala, we achieve the same optimization by custom `iterate` functions that compute multiple components at once. As an example, consider the computation of the residual component followed by the restrict component: Instead of computing the residual for the whole field first and then restrict the field produced by the residual, we compute the residual only for two rows and restrict the residual before the next rows are processed. This pipelined processing allows us to hold the result of the restrict component in cache on the CPU and to merge compute kernels on the GPU when using scratchpad (local or shared) memory. On the GPU, this has the same effect as loop fusion. Listing 10.16 illustrates this for the CPU. The index passed to the `residual` and `restrict` component refers to the temporary field. The offset to the current row of the other fields is tracked in the `Field` object and is used when accessing field elements. Merging the two components is only valid if the operation of the multigrid components is known: in our example, the restrict component is allowed to access two rows only. Otherwise, a larger temporary array has to be allocated and pre-computed before applying `restrict`.

```
fn vcycle(field: Field, lvls: int, vsteps: int, ssteps: int,
          smoother:    fn(/* ... */) -> (),
          residual:    fn(/* ... */) -> (),
          restrict:    fn(/* ... */) -> (),
          interpolate: fn(/* ... */) -> ()) -> Field {
  // allocate memory for all lvls: Sol, RHS, Res, Tmp
  fn vcycle_rec(lvl: int) -> () {
    if lvl == lvls-1 {
      for i in range(0, ssteps) { // solve by ssteps smooths
        if i>0 { swap(Sol(lvl), Tmp(lvl)); }
        for x, y in iterate(Sol(lvl)) {
          solver(x, y, /*fields*/);
        }
      }
    } else {
      for i in range(0, ssteps) {                // pre-smoothing
        if i>0 { swap(Sol(lvl), Tmp(lvl)); }
        for x, y in iterate(Sol(lvl)) {
          solver(x, y, /*fields*/);
        }
      }
      for x, y in iterate(Res(lvl)) {         // compute residual
        residual(x, y, /*fields*/);
      }
      for x, y in iterate(RHS(lvl+1)) {      // restrict residual
        restrict(x, y, /*fields*/);
      }
      vcycle_rec(lvl+1);                                  // recurse
      for x, y in iterate(Sol(lvl)) {   // interpolate error and
        interpolate(x, y, /*fields*/); // coarse grid correction
      }
      for i in range(0, ssteps) {               // post-smoothing
        if i>0 { swap(Sol(lvl), Tmp(lvl)); }
        for x, y in iterate(Sol(lvl)) {
          solver(x, y, /*fields*/);
        }
      }
    }
  }

  for i in range(0, vsteps) {
    vcycle_rec(0);
  }
}

let res = @vcycle(field, lvls, vsteps, ssteps,
                  jacobi, residual, restrict, interpolate);
```

Listing 10.14.: V-cycle implementation in Impala

if *coarsest level*
  solve $A^h u^h = f^h$ exactly or by many smoothing iterations
else
  $\bar{u}_h^{(k)} = \mathcal{S}_h^{\nu_1}\left(u_h^{(k)}, A^h, f^h\right)$ *pre-smoothing*
  $r^h = f^h - A^h \bar{u}_h^{(k)}$ *compute residual*
  $r^H = R r^h$ *restrict residual*
  $e^H = V_H\left(0, A^H, r^H, \nu_1, \nu_2\right)$ *recurse*
  $e^h = P e^H$ *interpolate error*
  $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e^h$ *coarse grid correction*
  $u_h^{(k+1)} = \mathcal{S}_h^{\nu_2}\left(\tilde{u}_h^{(k)}, A^h, f^h\right)$ *post-smoothing*
end

Listing 10.15.: V-cycle: $u_h^{(k+1)} = V_h\left(u_h^{(k)}, A^h, f^h, \nu_1, \nu_2\right)$.

```
fn iterate_rr(Sol: Field, Res: Field, RHSF: Field, RHSC: Field,
              residual: fn(/* ... */) -> (),
              restrict: fn(/* ... */) -> ()) -> () {
  let mut tmp: Field = { /* ... */ }; // temp array for 2 rows

  for y in $range_step(0, Res.rows, 2) {
    for yi in @range(0, 2) {
      for x in $range(0, Res.cols) { // residual for two rows
        @residual(x, yi /* ... */ Sol, tmp, RHSF);
      }
    }
    for x in $range(0, RHSC.cols) { // restrict the residual
      @restrict(x, 0 /* ... */ tmp, RHSC);
    }
  }
}
```

Listing 10.16.: Merging residual and restrict on the CPU

Results

While we have shown in Section 10.7.2 that we achieve competitive performance for stencil codes, the multigrid iteration offers further optimization opportunities when components are scheduled in a clever way. Figure 10.5 shows the speedup we get by merging the residual and restrict components for the first level of the V-cycle (smooth, residual, restrict, interpolate). The speedup is between 11 % on the CPU and up to 20 % on the GPU. Considering only the residual and restrict component, the computation is 25 % (27 %) faster on the CPU (AVX) and 42 % (45 %) faster on the GPU when using NVVM (SPIR). For AVX, we only vectorize the smooth and residual component. Vectorizing the restrict and interpolate components would be slow due to their incoherent memory access patterns. On the Iris 5100, the execution takes 16 % longer when the two components are merged. Note that this is expected since the scratchpad memory is mapped to slow global memory in the Iris 5100 architecture. Consequently, the specialization for the Iris 5100 would not make use of scratchpad memory.

Furthermore, we compare the performance of our specialized V-cycle implementation against the performance of generated implementations by HIPA^{cc}, a DSL framework for stencil computations [Mem+16]. HIPA^{cc} provides CUDA and OpenCL back ends for execution on GPUs. We use the HIPA^{cc} implementation from [Mem+14a], which uses the same V-cycle components as our implementation. For the first level of the V-cycle, our normal implementation has the same performance on the Iris 5100 (32.54 ms vs. 32.81 ms), is 8 % faster on the Radeon R9 290X (2.26 ms vs. 2.43 ms), and is 9 % slower on the GTX 680 (4.78 ms vs. 4.35 ms). Our merged implementation (see Figure 10.5) outperforms the HIPA^{cc} implementations on the Radeon R9 290X by 34 % and on the GTX 680 by 12 %.

Discussion

Our implementation can be easily extended to express different multigrid iterations. It is actually sufficient to change the recursion in the V-cycle implementation in order to get the schedule for the W-cycle multigrid iteration.

The evaluation has shown that we can map the same high-level description to different target platforms by providing target-specific mappings. Moreover, we merge multiple components as shown exemplarily for the

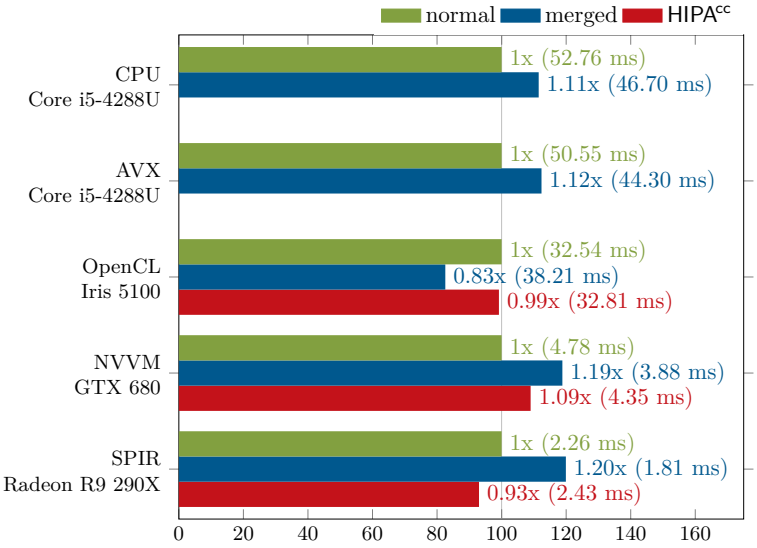


Figure 10.5.: Speedup from fusing the residual and restrict computation for the first level (4096×4096) of the V-cycle (smooth, residual, restrict, interpolate). The speedup over HIPA^{cc} implementations is also given where available.

residual and restrict components. This yields specialized implementations that outperform the implementations generated by HIPA^{cc} because HIPA^{cc} does not fuse the kernels.

A very large part of space-time must be investigated, if reliable results are to be obtained.

Alan Turing

11

Closure Elimination and Code Generation

This chapter leaves λ^{cps} behind and comes back to Thorin. We have already discussed how to translate Thorin to λ^{cps} and vice versa. This chapter now investigates how to optimize Thorin programs and how to generate low-level code for them.

To this end, we introduce a well-defined subset of Thorin programs called *control-flow form (CFF)*, which is akin to an SSA-based CFG. These programs do *not* need closure allocation. Furthermore, we define the set of *CFF-convertible* programs, a superset of CFF programs. This superset includes typical higher-order programming idioms like `map`, `fold`, and generators. With the help of *lower-to-CFF*, an aggressive closure elimination, Thorin transforms CFF-convertible programs into CFF programs (see Figure 11.1). The main ingredient for closure elimination is *lambda mangling*, a novel program transformation that partially in- and outlines functions.

Coming back to Listing 9.5, Listing 9.5c depicts the Thorin representation of the `range` function. This program is CFF-convertible. Applying lower-to-CFF generates Listing 9.5d, which is in CFF.

First, we discuss lambda mangling, then code generation and closure elimination. Afterwards, we evaluate the presented techniques in terms of the generated code's efficiency and lambda mangling's software-engineering complexity. Finally, we give an overview of related work. At that point, it will be easier to understand the differences to our own work.

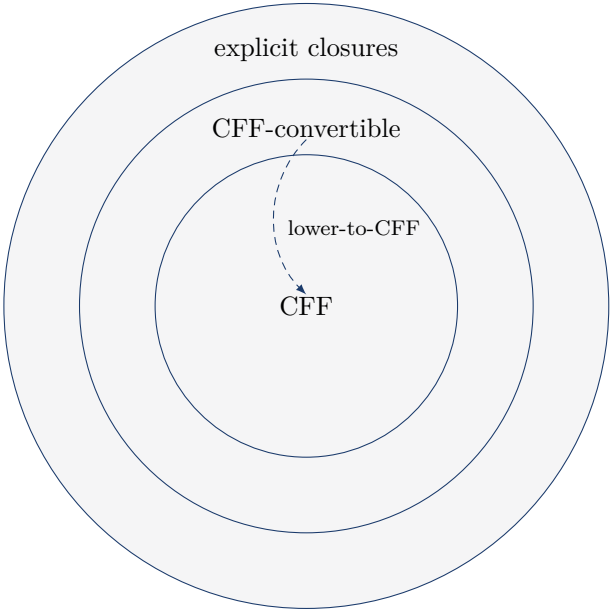


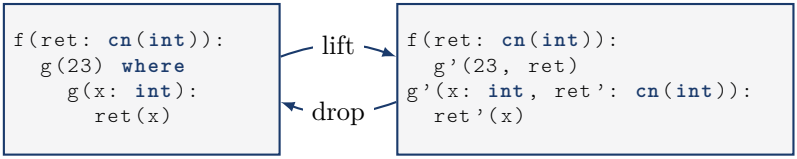
Figure 11.1.: Classes of Thorin programs

11.1. Lambda Mangling

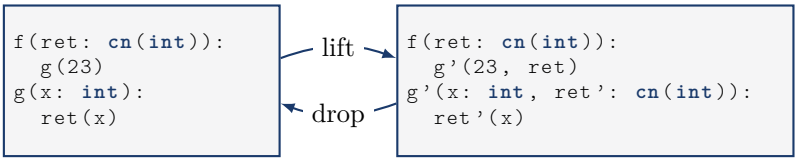
This section presents Thorin's main transformation primitive: *lambda mangling*—a combination of lambda lifting [Joh85] and dropping [DS00]. We demonstrate how many traditional compiler optimizations can be implemented with this transformation.In addition, the closure elimination algorithm in Section 11.2 resorts to lambda mangling.

When reading Figure 11.2a from left to right, we see how continuation `g` is *lambda-lifted* out of `f` by introducing a new parameter `ret'`, which eliminates `g`'s free variable `ret`. The new lifted version is then called `g'`. Likewise—when reading Figure 11.2a from right to left—we see how continuation `g'` is *lambda-dropped* into the body of `f` by eliminating `g'`'s parameter `ret'` and introducing a free variable `ret`.

Figure 11.2b depicts the same procedure for a Thorin program. Since Thorin programs are blockless, we do not have to move `g`/`g'` out of/into `f`.



(a) λ^{cps}



(b) Thorin

Figure 11.2.: Lambda lifting/dropping

The scope analysis (see Section 9.2.3) will identify g to be a continuation nested in f (in the left box) whereas g' and f will be discovered as two independent continuations (in the right box).

11.1.1. Combining Lambda Lifting and Dropping

The continuation pow(a, b) in Listing 11.1a computes a^b. The continuation has two call sites: in calcx and calcy. As both callers pass 3 for parameter b to pow, we would like to specialize pow. For this reason, we drop pow's parameter b and substitute each occurrence of b with 3. In doing so, we also apply local optimizations (constant propagation, common subexpression elimination, etc.). Thus, the check b = 0 and related blocks are eliminated in the new continuation $\texttt{pow}_d$ (see Listing 11.1b). Moreover, we update the call sites in calcx and calcy to call the new continuation $\texttt{pow}_d$ instead.

The scope of continuation pow in Listing 11.1a contains f's parameter ret. Thus, ret is a free variable in pow's scope. Suppose, we would like to eliminate pow's dependency on f. To this end, we lift pow by introducing a new parameter $\texttt{ret}_l$ (see Listing 11.1c).

We can apply both transformations by either first dropping and then lifting or first lifting and then droping (see Listing 11.1d). In order to reduce compile time, we can apply both transformations simultaneously. Both,

```
f(x: int, y: int,
  ret: cn(int)):
    br(/*...*/, calcx, calcy)
pow(a: int, b: int):
    br(b=0, then, else)
then():
    ret(1)
else():
    head(0, a)
head(i: int, r: int):
    br(i<b, body, next)
body():
    head(i+1, r*a)
next():
    ret(r)
calcx():
    pow(x, 3)
calcy():
    pow(y, 3)
```

(a) The nested `pow` computes a^b.

```
f(x: int, y: int,
  ret: cn(int)):
    br(/*...*/, calcx, calcy)
pow_d(a_d: int):
    head(0, a_d)




head(i: int, r: int):
    br(i<3, body, next)
body():
    head(i+1, r*a_d)
next():
    ret(r)
calcx():
    pow_d(x)
calcy():
    pow_d(y)
```

(b) Dropped $\texttt{pow}_d$ computes ${a_d}^3$.

```
pow_l(a_l: int, b_l: int,
      ret_l: cn(int)):
    br(b_l=0, then, else)
then():
    ret_l(1)
else():
    head(0, a_l)
head(i: int, r: int):
    br(i<b_l, body, next)
body():
    head(i+1, r*a_l)
next():
    ret_l(r)

f(x: int, y: int,
  ret: cn(int)):
    br(/*...*/, calcx, calcy)
calcx():
    pow_l(x, 3, ret)
calcy():
    pow_l(y, 3, ret)
```

(c) Lifted $\texttt{pow}_l$ doesn't use free variables.

```
pow_m(a_m: int, ret_m: cn(int)):
    head(0, a_m)




head(i: int, r: int):
    br(i<3, body, next)
body():
    head(i+1, r*a_m)
next():
    ret_m(r)

f(x: int, y: int,
  ret: cn(int)):
    br(/*...*/, calcx, calcy)
calcx():
    pow_m(x, ret)
calcy():
    pow_m(y, ret)
```

(d) Dropped and lifted $\texttt{pow}_m$

Listing 11.1.: Lambda Mangling

dropping and lifting, extend the program Φ (see Section 9.2.2) by a clone of a continuation's scope while rewriting all expressions with respect to a map $\mathcal{M}$. First, we have to define how to recursively rewrite an expression

$$\begin{aligned}
\mathcal{M}[\![e]\!] &= \mathcal{M}(e) && \text{if } e \in dom(\mathcal{M}) \\
\mathcal{M}[\![\ell]\!] &= \ell \\
\mathcal{M}[\![x_i]\!] &= x'_i && \text{where } \ell(\overline{x:t}) \text{ and } \mathcal{M}[\![\ell]\!](\overline{x':t'}) \\
\mathcal{M}[\![\boxtimes(e_1,\dots,e_n)]\!] &= \boxtimes(\mathcal{M}[\![e_1]\!],\dots,\mathcal{M}[\![e_n]\!])
\end{aligned}$$

and a body

$$\mathcal{M}[\![e_0(e_1,\dots,e_n)]\!] = \mathcal{M}[\![e_0]\!](\mathcal{M}[\![e_1]\!],\dots,\mathcal{M}[\![e_n]\!]) \ .$$

Reconsider Listing 11.1a as a starting point. In the case of lambda dropping (see Listing 11.1b), we substitute `b` with `3` and keep the parameter `a` called $\texttt{a}_d$ in the dropped version. We specify this mapping as follows:

$$\mathcal{M}_d \coloneqq \{\texttt{a} \mapsto \texttt{a}_d, \texttt{b} \mapsto 3\} \ .$$

In the case of lambda lifting (see Listing 11.1c), we substitute `ret` with $\texttt{pow}_l$'s new parameter $\texttt{ret}_l$ and keep parameters `a` and `b` called $\texttt{a}_l$ and $\texttt{b}_l$ in the lifted version:

$$\mathcal{M}_l \coloneqq \{\texttt{a} \mapsto \texttt{a}_l, \texttt{b} \mapsto \texttt{b}_l, \texttt{ret} \mapsto \texttt{ret}_l\} \ .$$

We can simultaneously drop `b` with `3` and lift (see Listing 11.1d) `pow`'s free variable `ret` to a new parameter $\texttt{ret}_m$ while obeying the mapping

$$\mathcal{M}_m \coloneqq \{\texttt{a} \mapsto \texttt{a}_m, \texttt{b} \mapsto 3, \texttt{ret} \mapsto \texttt{ret}_m\} \ .$$

The algorithm in Listing 11.2 performs this reconstruction for the whole scope of an entry label ℓ_e, with the new signature $\overline{x'_e : t'_e}$, and a corresponding mapping $\mathcal{M}$. In our example:

```
mangle(Φ, pow, (a_m: int, ret_m: cn(int)), M_m)
```

All newly created continuations are added to the program Φ.

11.1.2. Recursion

As all calls in CPS occur in tail position, it is tempting to think that only tail-recursion happens in a CPS program. This is not the case as can

function *mangle*$(\Phi, \ell_e, \overline{x'_e : t'_e}, \mathcal{M})$
  let $\ell_e(\overline{x_e : t_e}) : b_e \in \Phi$ ◁ *get* ℓ_e*'s signature and body*
  ◁ *for each continuation in scope except the entry*
  foreach $\big(\ell(\overline{x : t}) : b\big) \in \mathit{scope}_\Phi(\ell_e) \setminus \big(\ell_e(\overline{x_e : t_e}) : b_e\big)$ **do**
    $\mathcal{M}[\![\ell]\!] \leftarrow$ *fresh continuation name*
    foreach $x_i \in \overline{x}$ **do**
      $\mathcal{M}[\![x_i]\!] \leftarrow$ *fresh parameter name*
    end
  end
  ◁ *for each continuation in scope except the entry*
  foreach $\big(\ell(\overline{x : t}) : b\big) \in \mathit{scope}_\Phi(\ell_e) \setminus \big(\ell_e(\overline{x_e : t_e}) : b_e\big)$ **do**
    ◁ *rewrite continuation and add to program*
    $\Phi \leftarrow \Phi \cup \big\{\mathcal{M}[\![\ell]\!](\mathcal{M}[\![x_1]\!] : t_1, \ldots, \mathcal{M}[\![x_n]\!] : t_n) : \mathcal{M}[\![b]\!]\big\}$
  end
  ◁ *now deal with entry*
  $\ell'_e \leftarrow$ *fresh continuation name for new entry*
  ◁ *rewrite entry and add to program*
  $\Phi \leftarrow \Phi \cup \big\{\ell'(x'_{e_1} : t'_{e_1}, \ldots, x'_{e_n} : t'_{e_n}) : \mathcal{M}[\![b_e]\!]\big\}$
  return ℓ'_e ◁ *return entry to new mangled region*
end

Listing 11.2.: Lambda mangling expects the program Φ to work on, the entry label ℓ_e of the scope to mangle, the new signature $\overline{x'_e : t'_e}$, and a map $\mathcal{M}$ that maps ℓ_e's signature to the new scope.

```
fn fac(n: int) -> int {
    if n<=1 {

        1
    } else {
        fac(n-1) * n
    }
}
```

(a) Impala

```
fac(n: int, ret: cn(int)):
  br(n≤1, then, else)
then():
  ret(1)
else():
  fac(n-1, cont)
cont(res: int):
  ret(n*res)
```

(b) Thorin

Listing 11.3.: Naïve, recursive implementation of factorial

be seen in Listing 11.3: The continuation `else` passes `cont` to `fac`. The continuation `cont` evaluates to a closure. This closure captures the state of this iteration. For this reason, `fac` itself is not tail-recursive albeit the recursive call of `fac` occurs in `else`'s tail position.

Compare this to the tail-recursive implementation in Listing 11.4a. This version consists of a base case `fac` and a helper continuation `help` that implements the loop. The recursive call in `help` passes $\texttt{ret}_h$ to `help`. Parameter $\texttt{ret}_h$ is not a continuation abstraction but a parameter that is just passed around. Therefore, no closure has to be captured.

In conclusion, it is important to distinguish tail-recursion and recursive tail-calls; standard text book definitions for tail-recursion [e.g., HR99, chap. 17.4] do not apply to CPS programs. Thus, we introduce our own nomenclature that is based on the CFG obtained by a CFA (see Section 10.6).

Definition 11.1 (CFG Node Classification)
We say a call is

- *recursive* if it is part of a strongly connected component (SCC) within the CFG,
- *simply recursive* if it is *recursive* and within the scope of its callee,
- *mutually recursive* if it is *recursive* and *not simply recursive*, and
- *first-order recursive* if it is *recursive* and only uses *static parameters* (see below) as higher-order arguments.

We call

- a parameter that does not change its value within an SCC a *static parameter*, and
- an SCC that is only formed by *recursive* calls taking arguments of zeroth order a *loop*.

Example 11.1 (CFG Node Classification)
Reconsider Listing 11.4a. The call of `help` within `then′` is simply recursive since `then′` belongs to `help`'s scope. The call of `help` within `else` is *not* recursive as it is not part of the SCC formed by `help` and `then′`. The parameter $\texttt{ret}_h$ is static in that SCC. This makes the call of `help` in `then′` first-order recursive.

The call of `fac` within `else` in Listing 11.3b is *not* first-order recursive as the call passes the higher-order argument `cont`. Note how the definition of first-order recursion reflects non-CPS tail-recursion.

Mangling Simple Recursion

In the following, we describe how to perform several code transformation with the help of lambda mangling by means of Listing 11.4a.

Tail-Recursion Elimination. Let us drop `help` into `fac` by dropping `help`'s parameters $\texttt{n}_h$ and $\texttt{ret}_h$ with `fac`'s parameters `n` and $\texttt{ret}_f$. However, our algorithm would not touch the recursive call within `help`'s scope because we do not put the entry ℓ_e = `help` into the map $\mathcal{M}$. We would obtain a dropped version of `help`, say $\texttt{help}_d$, which still calls the original continuation `help`. Just substituting ℓ_e with the new mangled continuation would create an ill-typed call site:

```
help_d(i+1, r*i, n_h, ret_h)
```

But the recursive call

```
help(i+1, r*i, n_h, ret_h)
```

uses as third argument $\texttt{n}_h$ and fourth argument $\texttt{ret}_h$. These are `help`'s static parameters. For this reason, we replace this call with $\texttt{help}_d\texttt{(i+1, r*i)}$ in Listing 11.4b. Moreover, we replace the call site $\texttt{help(1, 2, n, ret}_f\texttt{)}$ in `else` nested inside `fac` with $\texttt{help}_d\texttt{(1, 2)}$ as holds:

```
help(1, 2, n, ret_f)  ≡  help(1, 2, n_h, ret_h)  ≡  help_d(1, 2)
```

Note that the resulting program is α-equivalent to the iterative implementation in Listing 9.4d. In other words, with lambda mangling we performed *tail-recursion elimination* by transforming `help` to a loop $\texttt{help}_d$.

Loop Peeling. When we drop all parameters of the resulting continuation $\texttt{help}_d$ with `1` and `2`, we perform *loop peeling* (see Listing 11.4c). In this case, we cannot substitute the recursive call

```
help_d(i+1, i*r)
```

by $\texttt{help}_p\texttt{()}$ as the arguments are not static parameters.

Loop Unrolling. Based on the program in Listing 11.4b, we can also drop all parameters of $\texttt{help}_d$ with `i+1` and `r*i`. This performs *loop unrolling* (see Listing 11.4d). For the same reason as above, we cannot substitute the recursive call

```
help_d(i+1, i*r)
```

by calling the new continuation $\texttt{help}_u\texttt{()}$, either.

Summary. In general, we substitute calls to the entry continuation by calls to the mangled one if all dropped parameters are static. We can simply add a check for this pattern in Listing 11.2 to perform this rewrite.

Mangling Mutual Recursion

Finally, we discuss an extension of lambda mangling that allows us to transform first-order, mutually recursive functions into a loop.

Mutual Tail-Recursion Elimination. In Listing 11.5a continuations `is_even` and `is_odd` invoke each other in a mutually recursive way. There exists only a single user from the outside: continuation `foo` calls `is_even`. For performance reasons, we would like to drop `is_even` and `is_odd` into `foo` by substituting $\texttt{ret}_e/\texttt{ret}_o$ with `foo`'s parameter `ret`. So far, our mangling algorithm does not support this transformation: When dropping `is_even` to $\texttt{is\_even}'$, we do not know that we are going to analogously drop `is_odd`, too, and thus, cannot substitute the recursive call

```
fac(n: int, ret_f: cn(int)):
  br(n≤1, then, else)
then():
  ret_f(1)
else():
  help(1, 2, n, ret_f)

help(i: int, r: int,
     n_h: int, ret_h: cn(int)):
  br(i≤n_h, then′, else′)
then′():
  help(i+1, r*i, n_h, ret_h)
else′():
  ret_h(r)
```

(a) Tail-recursive factorial

```
fac(n: int, ret_f: cn(int)):
  br(n≤1, then, else)
then():
  ret_f(1)
else():
  help_d(1, 2)

help_d(i: int, r: int):

  br(i≤n, then′_d, else′_d)
then′_d():
  help_d(i+1, r*i)
else′_d():
  ret_f(r)
```

(b) Tail-recursion elimination

```
fac(n: int, ret_f: cn(int)):
  br(n≤1, then, else)
then():
  ret_f(1)
else():
  help_p()

help_p():
  br(1≤n, then′_d, else′_d)
then′_p():
  help_d(2, 2)
else′_p():
  ret_f(2)

help_d(i: int, r: int):
  br(i≤n, then′_d, else′_d)
then′_d():
  help_d(i+1, r*i)
else′_d():
  ret_f(r)
```

(c) Loop peeling

```
fac(n: int, ret_f: cn(int)):
  br(n≤1, then, else)
then():
  ret_f(1)
else():
  help_d(1, 2)

help_d(i: int, r: int):
  br(i≤n, then′_d, else′_d)
then′_d():
  help_u()

help_u():
  br(i+1 ≤ n, then′_u, else′_u)
then′_u():
  help_d(i+2, r*i*i)
else′_u():
  ret_f(r*i)
else′_d():
  ret_f(r)
```

(d) Loop unrolling

Listing 11.4.: Various optimizations using lambda mangling

```
foo(i: int, ret: cn(bool)):
  is_even(i, ret)

is_even(i_e: int, ret_e: cn(bool)):
  br(i_e>0, then_e, else_e)
then_e():
  is_odd(i_e-1, ret_e)
else_e():
  ret_e(true)

is_odd(i_o: int, ret_o: cn(bool)):
  br(i_o>0, then_o, else_o)
then_o():
  is_even(i_o-1, ret_o)
else_o():
  ret_o(false)
```

(a) Functions `is_even` and `is_odd` are first-order recursive.

```
foo(i: int, ret: cn(bool)):
  is_even′(i)

is_even′(i′_e: int):
  br(i′_e>0, then′_e, else′_e)
then′_e():
  is_odd′(i′_e-1)
else′_e():
  ret(true)

is_odd′(i′_o: int):
  br(i′_o>0, then_e, else_o)
then′_o():
  is_even′(i′_o-1)
else′_o():
  ret(false)
```

(b) The optimized version consists of a loop.

Listing 11.5.: Lambda mangling to eliminate mutual tail-recursion formed by `is_even`, $\texttt{then}_e$, `is_odd`, and $\texttt{then}_o$.

```
is_odd(i_e-1, ret_e)
```

with

```
is_odd′(i′_e-1)
```

Likewise, we cannot replace the call

```
is_even(i_o-1, ret_o)
```

with

```
is_even′(i′_o-1)
```

when dropping `is_odd`.

However, we know beforehand the mapping

$$\mathcal{M}_e \coloneqq \{\texttt{i}_e \mapsto \texttt{i}'_e, \texttt{ret}_e \mapsto \texttt{ret}\}$$

when dropping `is_even` to `is_even′` and the mapping

$$\mathcal{M}_o \coloneqq \{\texttt{i}_o \mapsto \texttt{i}'_o, \texttt{ret}_o \mapsto \texttt{ret}\}$$

when dropping `is_odd` to `is_odd′`. Considering this during mangling, we can directly substitute any call of the form

```
is_even(X_e, ret_e)
```

with

```
is_even′(X_e)
```

and any call of the form

```
is_odd(X_o, ret_o)
```

with

```
is_odd′(X_o)
```

as both $\texttt{ret}_e$ and $\texttt{ret}_o$ are static parameters in the SCC formed by `is_even`, $\texttt{then}_e$, `is_odd`, and $\texttt{then}_o$. Finally, we obtain Listing 11.5b. Note that the recursive calls now form a loop.[1] Thus, we performed *mutual tail-recursion elimination.*

11.2. Code Generation

In order to translate Thorin programs to a lower-level program representation like machine code or an SSA-based representation (like LLVM), we classify Thorin continuations in the following way:

Definition 11.2 (Continuation Classification)
We say a continuation is

- *basic-block-like* if it is of first-order,
- *returning* if it is of second-order with exactly one first-order parameter,
- *top-level* if its scope does not contain free variables, and
- *bad* if it is neither basic-block-like, nor a top-level, returning continuation.

[1] This loop is by the way *irreducible* [see HU72] as it has two entries.

Example 11.2 (Continuation Classification)
Reconsider Listing 9.4. All basic blocks in the SSA-form version are basic-block-like continuations in Thorin. The continuation `fac` is returning and top-level.

Based on this classification we find the following definition:

Definition 11.3 (CFF)
We say a scope is in *CFF* if it does not contain bad continuations.

Example 11.3 (CFF)
As `fac`'s scope (see Listing 9.4) does not contain any bad continuations the scope is in CFF.

11.2.1. Converting CFF-Thorin to SSA Form

It is straightforward to generate code from a CFF program. For example, we can use Kelsey's algorithm [Kel95] to translate the program to SSA form:

- All returning continuations become ordinary continuations in the SSA-form program. The first-order parameter acts as "return".
- All basic-block-like continuations become basic blocks. For each parameter we introduce a ϕ-function. The corresponding arguments of the basic-block's predecessors determine the ϕ-function's arguments.
- Calls to returning continuations become "normal" calls. The parameter of the call's continuation forms the result value in the SSA-form program.

Example 11.4 (Convert CFF to SSA form)
Using this algorithm we obtain Listing 9.4b from Listing 9.4d.

```
function lower-to-CFF(Φ)
    foreach bad continuation ℓ reachable from main do
        foreach u ∈ uses(ℓ) do
            if u calls ℓ
                ℓ_d ← drop all higher-order arguments using
                  Listing 11.2
                rewrite u to call ℓ_d instead;
            end
        end
    end
end
```

Listing 11.6.: Lower-to-CFF: this algorithm eliminates bad continuations.

11.2.2. Closure Elimination

The remaining top-level continuations that are not in CFF can be translated with standard code generation techniques for higher-order CPS programs [Ste78; Kra+86; App06]. An alternative is to eliminate bad continuations as the remainder of this section discusses.

Suppose `main` is a special continuation within Φ that is axiomatically *reachable*. All continuations *not* reachable from `main` are *unreachable* and can be removed (see Section 10.6). Now, the idea is to specialize (via *mangle* from Listing 11.2) all higher-order arguments in all reachable calls of bad continuations (see Listing 11.6). This routine will only terminate if all bad continuations become unreachable. In general, it is undecidable whether this will be the case for an arbitrary program because the algorithm is yet another partial evaluation strategy that can also induce divergence (see Example 11.6). Nevertheless, the lowering algorithm always reduces non-recursive calls of bad continuations since each specialized call removes one use of the bad continuation. This property still holds when mangling first-order recursive continuations because the mangled version does not reference the original continuation anymore (see Section 11.1.2). When mangling recursive but not first-order recursive continuations, those references usually stay there. In this case, specializing the call does not decrease the number of uses of the bad continuation. Only if some other optimizations are triggered, these references may disappear.

```
foo(n: int, fret: cn()):
  range'(0, n)
range'(a': int, b': int):
  br(a'<b', then', else')
then'():
  lambda(a', cont')
cont'():
  range'(a'+1, b')
else'():
  next()
lambda(i: int, out: cn()):
  use(i, n, out)
next():
  fret()
```

(a) Mangling `range`

```
foo(n: int, fret: cn()):
  range'(0, n)
range'(a': int, b': int):
  br(a'<b', then', else')
then'():
  lambda'(a')
cont'():
  range'(a'+1, b')
else'():
  next()
lambda'(i': int):
  use(i', n, cont')
next():
  fret()
```

(b) Mangling `lambda`

Listing 11.7.: Lower-to-CFF optimizes the program in Listing 9.5c.

Definition 11.4 (CFF-Convertible)
We call a program *CFF-convertible* if all bad continuations are either non-recursive or first-order-recursive.

Example 11.5 (Closure Elimination)
Listing 9.5c invokes the higher-order continuation `range`. We now apply lower-to-CFF. First, lower-to-CFF mangles `range` to `range'` because `range` is a bad continuation with the higher-order parameters `yield` and `ret`. Note how mangling eliminates the static parameters (see Listing 11.7a). The returning continuation `lambda` with the higher-order parameter `out` is bad since it is nested inside of `f` due to its dependency on `f`'s parameter `n`. For this reason, lower-to-CFF mangles `lambda` and renders the program in shown Listing 11.7b, which is in CFF, and the algorithm terminates. Using lambda mangling, we also inline `lambda'` and specialize `range'`'s parameter `b'` to `n` to finally obtain Listing 9.5d.

Example 11.6 (Endless Mangling)
The following program is *not* first-order recursive because `f`'s parameters `ret1` and `ret2` are not static:

```
f(a: int, ret1: cn(int), ret2: cn(int)): f(a+1, R, S)
R(i: int): ret2(a+i)
S(j: int): ret1(a+j)

main(i: int, ret: cn(int)): f(i, ret, ret)
```

Thus, naïvely invoking lower-to-CFF will not terminate. This program must be closure-converted and compiled with traditional compilation techniques for higher-order continuations.

11.2.3. Enhancements

The presented lowering algorithm always uses lambda dropping. Nevertheless, in certain cases it is worthwhile to use lambda lifting instead as lifting will not increase the code size. However, lambda lifting is only reasonable in the case of returning, non-top-level continuations that only contain free variables of zeroth order (`lambda` in Listing 11.7a, for example). Then, the lifted continuation will be a returning, top-level continuation. Otherwise, when lifting free variables of order one or higher, the resulting continuation will not be a returning continuation anymore.

Moreover, lower-to-CFF always specializes *all* higher-order arguments. But in order to use Kelsey's algorithm, it suffices to have top-level, returning continuations. This means that it is fine to keep *one* first-order, potentially non-static parameter (the "return") of top-level, bad continuations. The property of first-order recursion must then only hold for the remaining parameters. This would allow us to transform a greater class of programs to CFF.

11.2.4. Summary

Compilation works as follows:

1. Identify all bad continuations.

2. Use lower-to-CFF to eliminate all non-recursive and first-order recursive uses. For such uses, lower-to-CFF will terminate. If all uses are of these kinds, the original bad continuation will become unreachable and can be removed.

	C	Impala	Rust	GHC
aobench	1.220	1.357	n/a	22.540
fannkuch-redux	27.137	28.070	n/a	34.670
fasta	2.313	1.517	n/a	1.443
mandelbrot	2.143	2.113	n/a	2.013
meteor-contest	0.047	0.043	0.050	0.327
n-body	5.497	6.130	5.163	6.867
pidigits	0.710	0.763	4.940	0.903
regex	6.477	6.470	18.020	7.720
reverse-complement	1.090	1.220	n/a	1.300
spectral-norm	4.423	4.480	n/a	19.347

Figure 11.3.: Median execution times in seconds (lower is better)

3. Translate all continuations that are not bad with Kelsey's [Kel95] algorithm.

4. Translate remaining continuations with conservative code generation techniques (closure conversion).

11.3. Evaluation

This section evaluates Thorin. We have already argued that Thorin is more expressive than classic SSA-based representations because Thorin supports higher-order functions. To begin with, we want to examine whether we have to pay for this expressiveness with less efficient code. Then, we investigate whether this expressiveness complicates the implementation of code transformations.

11.3.1. Performance

The Impala[2] compiler translates the source program to Thorin. Partial evaluation (see Chapter 10), closure elimination (see Section 11.2.2), and other standard optimizations are performed on top of this IR. Finally, Thorin either translates to C/CUDA/OpenCL or LLVM/SPIR/NVVM (see

[2] see `http://anydsl.github.io`

also Sections 10.7 and 11.2.1). Beside the semantic analysis, the Impala compiler does not perform any further analyses or transformations on the AST. In particular, Impala directly translates higher-order functions to Thorin in a straightforward manner.

In order to evaluate the effectiveness of our approach, we ported *The Computer Benchmark Game*[3] to Impala. We elided some programs that focus on measuring API or runtime overhead. Additionally, we ported `aobench`[4]. Figure 11.3 shows the median execution times of eleven runs for C, Impala, Rust (if available), and GHC in seconds. Our benchmark ran on an Intel® Ivy Bridge Core™ i7-3770K CPU. We consider the C implementations as baseline. We included Rust in our measurements as Impala has a similar syntax. We also included Haskell versions of the benchmarks in order to see how well GHC's Core IR performs. From the original benchmark suite, we selected programs that were neither hand-vectorized nor hand-parallelized. We used clang 3.4.2, rustc 0.11 and GHC 7.8.3. The exact compile flags and benchmarks are apparent in our benchmark suite.[5]

Although we used the C versions of the benchmarks as a template for the Impala version, it is important to note that Impala does not offer C-style `for`-loops. Instead, we use higher-order functions to write appropriate generators and rely on Thorin's lower-to-CFF phase. For example, in order to iterate over an interval, we use the higher-order function `range`.

To sum up, the performance of the Impala programs is mostly on a par with the C implementations except for `fasta` where Impala is about 1.5 times faster than C. Rust's performance depends on the quality of the libraries used under the hood. GHC is roughly on a par with C/Impala. However, some benchmarks—in particular `aobench`—run significantly slower.

We conclude that Thorin's expressiveness does not impact the quality of the generated code. In particular, a C-like implementation will also give the performance of a C implementation.

	SLoC	Volume	Difficulty	Effort
`CloneFunction.cpp`	359	21298	107	2269693
`CodeExtractor.cpp`	523	34599	124	4287983
`InlineFunction.cpp`	526	32288	109	3511359
`LoopUnroll.cpp`	279	15393	80	1229623
total	1687	120421	207	24968883
`mangle.cpp`	132	6636	75	496757

Figure 11.4.: Source lines of code (SLoC) and Halstead numbers for LLVM's C++ implementations compared to Thorin's mangle implementation

11.3.2. Engineering Effort

In order to estimate the engineering effort to develop code transformations, we compare the Halstead metric [Hal77] of Thorin's lambda mangling implementation (see Section 11.1) versus LLVM 3.4.2 (see Figure 11.4).[6] On the one hand, LLVM is a full featured compiler suite which biases these metrics towards Thorin. On the other hand, lambda mangling is much more versatile: For LLVM we did not include any source code to eliminate tail-recursion (see Section 11.1.2). Furthermore, LLVM completely lacks functionality for *partial* inlining or *partial* outlining and cannot represent higher-order functions at all. Still, LLVM's pendants are in total roughly 2.8 times more difficult to implement while they take about 50 times longer to program.

[3] see `http://benchmarksgame.alioth.debian.org/`
[4] see `https://code.google.com/p/aobench`
[5] see `https://github.com/AnyDSL/benchmarks-impala`
[6] SLoC were generated using David A. Wheeler's 'SLOCCount'; Halstead numbers were computed with `c3ms` [GF10].

11.4. Related Work

11.4.1. Lambda Lifting and Dropping

Lambda lifting was invented by Johnsson [Joh85]. His algorithm uses currying in order to abstract free variables:

$$\lambda x.\ x + y \quad \Rightarrow \quad \lambda y.\lambda x.\ x + y\ .$$

Danvy and Schultz [DS00] observe that we can directly append y to the parameter list in the case of first-order programs with multi-ary functions:

$$\lambda(x).\ x + y \quad \Rightarrow \quad \lambda(x, y).\ x + y\ .$$

Danvy and Schultz invented lambda dropping as a reverse transformation to Johnsson's algorithm. When dealing with higher-order functions, their algorithm uses Johnsson's currying approach since in general a compiler cannot rewrite call sites of a function passed as argument to another function. As we want to eliminate closures, currying is not an option for us. After all, a curried function is not a returning function anymore (see Definition 11.2). For this reason, lambda mangling does not use currying to introduce or eliminate parameters. It rather produces one new generalized and/or specialized function with an updated signature, which is at that point not connected to the rest of the program. It is in the responsibility of other passes to orchestrate mangling in a reasonable way and to connect a mangled function to the rest of the program properly. Due to Thorin's blockless representation, we can fuse both algorithms into one simple, recursive rewrite algorithm, which does not need the block floating/sinking pass of the original algorithms.

11.4.2. Static Argument Transformation

We are not the first to note that an argument/parameter pair of a recursive call is superfluous if the argument is just the parameter. The *static argument transformation* [San95] identifies such recursive calls and eliminates them.

Example 11.7 (Static Argument Transformation)
Consider Listing 11.8. Note that `f` itself is not recursive anymore and, hence, is a potential candidate for inlining.

```
f: (a, b)
    ... use(a) ...
    f(a, x)
```

(a) before

```
f: (a, b)
    letrec f': (b)
        ... use(a) ...
        f'(x)
    in  f'(b)
```

(b) after

Listing 11.8.: Static argument transformation

For mutually recursive functions, we must find out which parameters do not change their values within an SCC. For this reason, we rather speak of static *parameters* than of *arguments*. However, applying the static argument transformation on a mutual recursive function still leaves the function recursive whereby further inlining becomes problematic. Our algorithm on the other hand considers all functions within one SCC simultaneously and, thus, can eliminate static parameters (see Section 11.1.2).

11.4.3. Super-β Inlining

Super-β inlining [see Shi91, chap. 10] enables inlining of a closure call c with a function literal f. This transformation is only valid if

1. all applications of c are closures over f and
2. the environment at c is always equivalent to the one where the closure is captured.

A CFA (see Section 10.6) addresses the first requirement. Δ-CFA [MS06] also takes into account the environment where a closure is captured. This allows aggressive inlining and, hence, aggressive closure elimination.

The focus in our work is different: The presented lower-to-CFF algorithm (see Section 11.2.2) specializes higher-order functions even more aggressively (in contrast to inlining):

Example 11.8
Consider the function `range` from Listing 9.5. Suppose, there are two call sites of `range` that pass different function literals `g1` and `g2` to `range`'s parameter `f`:

```
range(..., g1, ...)
range(..., g2, ...)
```

A Δ-CFA discovers that inlining the closure call of `f` within `range`'s scope is not possible since condition (1) is violated. Thorin's lower-to-CFF on the other hand, specializes each call site and, thus, gets rid of the higher-order function `range` (see Section 11.2.2).

11.4.4. Other Closure-Elimination Strategies

The SML/NJ compiler [App06] allocates garbage-collected closures on the heap. An additional phase tries to find closures whose lifetime can be statically proven to be nested in a simple way. These closures can be allocated on the stack instead. Additionally, the compiler lambda-lifts functions that are not passed as argument to other functions. Then, these so-called *known* functions (as opposed to *escaping* functions) do not require a closure at all. An additional η-split phase splits functions that are used in a known as well as in an escaping context. Finally, SML/NJ's inliner tries to decrease *escaping* contexts. Other compilers like Scala[7] also rely on inlining to eliminate explicit closures.

In summary, state-of-the-art functional compilers rely on *inline heuristics* to eliminate closures. Thorin *guarantees* to eliminate all closures in CFF-convertible programs.

[7] see `http://magarciaepfl.github.io/scala`

Don't adventures ever have an end? I suppose not. Someone else always has to carry on the story.

J.R.R. Tolkien, The Lord of the Rings

12 Conclusions

The AnyDSL framework presented in this part of the thesis allows a DSL designer to embed her language *as a library* in the host language Impala; she does not need to implement or modify a compiler (as in Part I). The application programmer, on the other hand, only sees the DSL interface, which is just a usual library. Using this interface, he can write concise, readable, and maintainable code. Furthermore, with the help of higher-order functions, the DSL designer can abstract from hardware details and algorithmic variants. Experts for different machines can provide different implementations in order to effectively map the DSL to these machines.

This paradigm allows a clean separation between the application developer, the DSL designer, and the machine expert. Using partial evaluation and an aggressive closure elimination, all aspects are woven together. We demonstrate on two example DSLs that the resulting code even outperforms code that has been hand-optimized for various architectures. In addition, Pérard-Gayot et al. [Pér+17] show similar results for ray traversal (the core of a ray tracer).

AnyDSL's IR Thorin blends concepts from functional, classic SSA-based, and modern graph-based representations. Therefore, Thorin is very expressive and equally well-suited for imperative as well as functional programs. At the same time, we show that code transformations are in fact simpler to implement on Thorin than on classic SSA-based representations because Thorin knows fewer concepts: every aspect of control flow is uniformly represented by continuations.

The partial evaluation strategy presented in Chapter 10 might not be the end of the line. But for *any* evaluation strategy (including partial ones) that is a subset of the indeterministic reduction system presented in Section 9.3 preservation (see Lemma 9.2) and confluence (see Theorem 10.2) hold. This builds a profound basis for future work.

Appendices

Noli turbare circulos meos!

presumably Archimedes

A Notation

This chapter briefly discusses the notation used throughout this thesis.

A.1. Functions and Sequences

Let f be a function, then $dom(f)$ denotes the domain and $codom(f)$ the codomain of f. We write $f[x \mapsto y]$ to express a function that is identical to f except that we add or update the mapping $x \mapsto y$:

$$f[x \mapsto y] : dom(f) \cup \{x\} \to codom(f) \cup \{y\}$$
$$x' \mapsto \begin{cases} y & \text{if } x' = x \\ f(x') & \text{otherwise.} \end{cases}$$

We write a sequence as $a_1, \dots, a_n =: \overline{a}$ while $\varnothing$ denotes the empty sequence although it is sometimes just omitted. We write $a \in A$ if a occurs in the sequence A and use $\overline{a} \in A$ as shorthand for $\forall a_i \in A : a_i \in A$. The comma operator performs concatenation. Thus, "A, B" concatenates two sequences A and B whereas "A, a" adds the element a to the sequence A. We will often write $a : b$ to designate a pair if we want to stress that b is associated to a. We also view a sequence of pairs

$$A \coloneqq a_1 : b_1, \dots, a_n : b_n =: \overline{a : b}$$

in which each a_i is unique as bijective function:

$$A : \{a_1, \dots, a_n\} \to \{b_1, \dots, b_n\}$$
$$a_1 \mapsto b_1$$
$$\dots$$
$$a_n \mapsto b_n$$

This is, for example, the case for typing environments or states as we require all variable names to be unique. Following these definitions, we find

$$\begin{aligned} dom(A) &= \{a_1, \dots, a_n\} \\ codom(A) &= \{b_1, \dots, b_n\} \\ A[x \mapsto y] &= \overline{a : b}, x : y \\ A[a_1 \mapsto y] &= a_1 : y, a_2 : b_2, \dots, a_n : b_n \ . \end{aligned}$$

A.2. Relations

Definition A.1
Let $\rightharpoonup$ be a binary relation. The following constructors define its reflexive transitive closure $\rightharpoonup^*$:

$$\text{Id}\ \frac{a \rightharpoonup b}{a \rightharpoonup^* b} \qquad \text{Refl}\ \frac{}{a \rightharpoonup^* a} \qquad \text{Trans}\ \frac{a \rightharpoonup^* b \qquad b \rightharpoonup^* c}{a \rightharpoonup^* c} \ .$$

Definition A.2 (Normal form)
Let $\rightharpoonup$ be a binary relation on a set A. We say $a \in A$ is in *normal form* if there is no $a' \neq a$ such that $a \rightharpoonup a'$. We write $a\!\nmid$ as shorthand if we require a to be in normal form. Likewise, we write $a\!\downharpoonright$ if we require a to be *not* in normal form.

> *Remark.* This definition also works for a reflexive relation—say $\leftrightarrow$. Although $a \leftrightarrow a$ holds for all a, $\hat{a}\!\updownarrow$ denotes an element $\hat{a}$ that cannot be stepped into another element *different* from $\hat{a}$.

A.3. Syntax

In this thesis we use Backus-Naur [Bac59] form to describe the *abstract syntax* of a language. *Expanded syntax* is a subset of the abstract syntax and appears grayed out in the presentation. Expanded syntax is not directly accessible to the programmer. It solely materializes internally, for instance, in typing or evaluation rules. The *concrete syntax* extends the abstract syntax by parentheses, for example, in order to disambiguate expression nesting. Additionally, we sometimes define *syntactic sugar*: This syntax

directly translates to a more verbose concrete syntax and is merely used to make examples "sweeter" to read.

The abstract syntax of some language $\mathcal{L}$ induces a tree. Let $a, b \in \mathcal{L}$. We write $a \preceq b$ to denote that a is a subtree of b. The trees a and b may coincide.

There is only an idea. And ideas are bulletproof.
V for Vendetta

B Full Proofs

B.1. Imp

Lemma 4.3 (Imp: Progress)
Every Imp *term is either a term value or can be stepped into another term. To be more precise: Let* $\vdash \Phi$ *and* $\Gamma \trianglerighteq \sigma$.

$$\textit{If} \quad \begin{array}{l} \Phi;\Gamma \vdash s : t \\ \Phi;\Gamma \vdash e : t \end{array},$$

$$\textit{then} \quad \begin{array}{l} s = \texttt{skip} \\ \quad \vee \\ e = \nu_t \end{array} \quad \vee \quad \begin{array}{l} s = \texttt{return } \nu_t\texttt{;} \end{array} \quad \textit{or} \quad \begin{array}{l} \exists s', \sigma' : \Phi \vdash \sigma; s \to \sigma'; s' \\ \exists e', \sigma' : \Phi \vdash \sigma; e \to \sigma'; e' \end{array}.$$

Proof. By mutual induction on a derivation of $\begin{array}{l} \Phi;\Gamma \vdash s : t \\ \Phi;\Gamma \vdash e : t \end{array}$.

Case:

$$\text{TS-Skip}\ \frac{}{\Phi;\Gamma \vdash \texttt{skip} : \bot}$$

This is the final configuration.

Case:

$$\text{TS-Ret}\ \frac{\Phi;\Gamma \vdash e : t}{\Phi;\Gamma \vdash \texttt{return } e\texttt{;} : t}$$

Using the induction hypothesis we deduce:

`return` ν_t:	This is the final configuration.
otherwise:	E-Eval applies.

Case: $\text{TS-Expr}\ \dfrac{\Phi;\Gamma \vdash e : t_e \qquad \Phi;\Gamma \vdash s : t}{\Phi;\Gamma \vdash e;\ s : t}$

Using the induction hypothesis we deduce:

$\nu_t;\ s$:	ES-Expr applies.
otherwise:	E-Eval applies.

Case: $\text{TS-Decl}\ \dfrac{\Phi;\Gamma[x \mapsto t_x] \vdash s : t}{\Phi;\Gamma \vdash t_x\ x;\ s : t}$

Axiomatically, ES-Decl applies.

Case: $\text{TS-Assign}\ \dfrac{x : t \in \Gamma \qquad \Phi;\Gamma \vdash e : t_e \qquad \Phi;\Gamma \vdash s : t}{\Phi;\Gamma \vdash x = e;\ s : t}$

$e = \nu_t$:	Since $x : t_e \in \Gamma$ and $\Gamma \backsimeq \sigma$ we know $x : \nu'_{t_e} \in \sigma$ for some ν'_{t_e}. Thus, ES-Assign applies.
otherwise:	E-Eval applies.

Case: $\text{TS-If}\ \dfrac{\Phi;\Gamma \vdash e : \texttt{bool} \qquad \Phi;\Gamma \vdash s_t : \bot \qquad \Phi;\Gamma \vdash s_f : \bot \qquad \Phi;\Gamma \vdash s_r : t}{\Phi;\Gamma \vdash \texttt{if}\ (e)\ s_t\ \texttt{else}\ s_f\ s_r : t}$

Using the induction hypothesis we deduce:

$e = \nu_{\texttt{true}}$:	ES-IfT applies.
$e = \nu_{\texttt{false}}$:	ES-IfF applies.
otherwise:	E-Eval applies.

Case: $\text{TS-While}\ \dfrac{\Phi;\Gamma \vdash e : \texttt{bool} \qquad \Phi;\Gamma \vdash s_b : \bot \qquad \Phi;\Gamma \vdash s_r : t}{\Phi;\Gamma \vdash \texttt{while}\ (e)\ s_b\ s_r : t}$

Axiomatically, ES-While applies.

Case: $\text{TE-Var}\ \dfrac{x : t \in \Gamma}{\Phi;\Gamma \vdash x : t}$

Since $x : t \in \Gamma$ and $\Gamma \trianglerighteq \sigma$ we know $x : \nu_t \in \sigma$ for some ν_t. Thus, EE-Var applies.

Case: $\text{TE-Val}\ \dfrac{}{\Phi;\Gamma \vdash \nu_t : t}$

This is the final configuration.

Case: $\text{TE-Call}\ \dfrac{t\ \ell(t_1\ x_1, \ldots, t_n\ x_n)\ \{\ s\ \} \in \Phi \qquad \Phi;\Gamma \vdash e_1 : t_1 \qquad \cdots \qquad \Phi;\Gamma \vdash e_n : t_n}{\Phi;\Gamma \vdash \ell(e_1, \ldots, e_n) : t}$

Using the induction hypothesis we deduce:

$\overline{e} = \overline{\nu_t}$:	Hence, EE-Call applies.
otherwise:	E-Eval applies.

Case: $\text{TE-Stmt}\ \dfrac{\Phi;\Gamma \vdash s : t}{\Phi;\Gamma \vdash \{\ s\ \}_\sigma : t}$

Using the induction hypothesis we deduce:

$s = \texttt{return}\ \nu_t;$:	EE-Stmt applies.
otherwise:	E-Eval applies.

□

Lemma 4.4 (IMP: Preservation)
If a well-typed IMP *term takes a step of evaluation, the resulting term is*

also well-typed. To be more precise: Let $\vdash \Phi$, $\Gamma \preceq \sigma$, *and* $\Gamma' \preceq \sigma'$.

$$\textit{If} \quad \begin{array}{l} \Phi;\Gamma \vdash s : t \\ \Phi;\Gamma \vdash e : t \end{array} \quad \textit{and} \quad \begin{array}{l} \Phi \vdash \sigma; s \to \sigma'; s' \\ \Phi \vdash \sigma; e \to \sigma'; e' \end{array}, \quad \textit{then} \quad \begin{array}{l} \Phi;\Gamma' \vdash s' : t \\ \Phi;\Gamma' \vdash e' : t \end{array}.$$

Proof. By mutual induction on a derivation of $\begin{array}{l} \Phi \vdash \sigma; s \to \sigma'; s' \\ \Phi \vdash \sigma; e \to \sigma'; e' \end{array}$. Many evaluation rules do not produce a new state. Thus, $\sigma' = \sigma$ in these cases. By Lemma 4.1 this means that $\Gamma' = \Gamma$. Many case distinctions will silently make use of this insight.

Case:

$$\text{E-Eval}\ \frac{\Phi \vdash \sigma; \hat{a} \to \sigma'; \hat{a}'}{\Phi \vdash \sigma; \underbrace{\mathcal{E}[\hat{a}]}_{a} \to \sigma'; \underbrace{\mathcal{E}[\hat{a}']}_{a'}}$$

By the induction hypothesis we know

$$\Phi;\Gamma' \vdash \hat{a}' : t \quad \text{where} \quad \Gamma' \preceq \sigma' .$$

Using the appropriate typing rule we find

$$\Phi;\Gamma' \vdash \underbrace{\mathcal{E}[\hat{a}']}_{a'} : t$$

as expected:

$$\underbrace{\mathcal{E}[\hat{a}]}_{a} = \underbrace{\texttt{return}\ \hat{a};}_{a} \qquad \text{TS-Ret}\ \frac{\Phi;\Gamma' \vdash \hat{a}' : t}{\Phi;\Gamma' \vdash \underbrace{\texttt{return}\ \hat{a}';}_{a'} : t}$$

$$\underbrace{\mathcal{E}[\hat{a}]}_{a} = \underbrace{\hat{a};\ s}_{a} \qquad \text{TS-Expr}\ \frac{\Phi;\Gamma' \vdash \overset{a'}{\hat{a}'} : t_a \qquad \Phi;\Gamma' \vdash s : t}{\Phi;\Gamma' \vdash \underbrace{\hat{a}';\ s}_{a'} : t}$$

$$\underbrace{\mathcal{E}[\hat{a}]}_{a} = \underbrace{x = \hat{a};\ s}_{a} \qquad \text{TS-Assign}\ \frac{x : t \in \Gamma' \qquad \Phi;\Gamma' \vdash \overset{a'}{\hat{a}'} : t_a \qquad \Phi;\Gamma' \vdash s : t}{\Phi;\Gamma' \vdash \underbrace{x = \hat{a}';\ s}_{a'} : t}$$

$$\underbrace{\mathcal{E}[\hat{a}]}_{a} = \underbrace{\texttt{if}\ (\hat{a})\ s_t\ \texttt{else}\ s_f\ s_r}_{a} \qquad \text{TS-If}\ \frac{\begin{array}{ll} \Phi;\Gamma' \vdash \hat{a}' : \texttt{bool} & \Phi;\Gamma' \vdash s_t : \bot \\ \Phi;\Gamma' \vdash s_f : \bot & \Phi;\Gamma' \vdash s_r : t \end{array}}{\Phi;\Gamma \vdash \underbrace{\texttt{if}\ (\hat{a}')\ s_t\ \texttt{else}\ s_f\ s_r}_{a'} : t}$$

$$\underbrace{\mathcal{E}[\hat{a}]}_{a} = \underbrace{\ell(\overline{\nu_t}, \hat{a}, \overline{e})}_{a} \qquad \text{TE-Call}\ \frac{t\ \ell(t_1\ x_1, \ldots, t_a\ x_a, \ldots, t_n\ x_n)\ \{\ s\ \} \in \Phi \quad \Phi;\Gamma' \vdash v_1 : t_1 \quad \cdots \quad \Phi;\Gamma' \vdash \hat{a}' : t \quad \cdots \quad \Phi;\Gamma' \vdash e_n : t_n}{\Phi;\Gamma \vdash \underbrace{\ell(v_1, \ldots, \hat{a}', \ldots, e_n)}_{a'} : t}$$

$$\underbrace{\mathcal{E}[\hat{a}]}_{a} = \underbrace{\{\ a\ \}_{\hat{\sigma}}}_{a} \qquad \text{TE-Stmt}\ \frac{\Phi;\Gamma' \vdash \hat{a}' : t}{\Phi;\Gamma' \vdash \underbrace{\{\ \hat{a}'\ \}_{\hat{\sigma}}}_{a'} : t}$$

Case:

$$\text{ES-Expr}\ \frac{}{\Phi \vdash \sigma; \nu_t;\ s \to \sigma; s}$$

Straightforward from the induction hypothesis.

Case:

$$\text{ES-Decl}\ \frac{}{\Phi \vdash \sigma; \underbrace{t_x\ x;\ \hat{s}}_{s} \to \sigma[x \mapsto 0_{t_x}]; \underbrace{\hat{s}}_{s'}}$$

We know:

$$\Gamma \trianglerighteq \sigma \quad \text{and} \quad \Gamma[x \mapsto t_x] = \Gamma' \trianglerighteq \sigma' = \sigma[x \mapsto 0_{t_x}]\ .$$

Thus, by the induction hypothesis:

$$\Phi;\Gamma' \vdash s' : t\ .$$

Case:

$$\text{ES-Assign}\ \frac{x : \nu'_t \in \sigma}{\Phi \vdash \sigma; x = \nu_{t_x};\ s \to \sigma[x \mapsto \nu_{t_x}]; s}$$

Similar to the previous case.

Case:

$$\text{ES-IfT}\ \frac{}{\Phi \vdash \sigma; \texttt{if (true)}\ s_t\ \texttt{else}\ s_f\ s_r \to \sigma; s_t \circ s_r}$$

By the induction hypothesis we know

$$\Phi;\Gamma \vdash s_t : \bot \quad \text{and} \quad \Phi;\Gamma \vdash s_r : t\ .$$

By Lemma 4.2 we conclude:

$$\Phi;\Gamma \vdash \underbrace{s_t \circ s_r}_{s'} : t\ .$$

Case: $\text{ES-IfF}\ \dfrac{}{\Phi \vdash \sigma; \texttt{if (false)}\ s_t\ \texttt{else}\ s_f\ s_r \to \sigma; s_f \circ s_r}$

Dual to the previous case.

Case: $\text{ES-While}\ \dfrac{}{\Phi \vdash \sigma; \underbrace{\texttt{while}\ (e)\ s_b\ s_r}_{s} \to \sigma; \underbrace{\texttt{if}\ (e)\ \{\ s_b\ \texttt{while}\ (e)\ s_b\ \}\ s_r}_{s'}}$

Using the induction hypothesis and having the syntactic sugar in mind (see Section 4.1.1) we derive:

$$\text{TS-If}\ \frac{\Phi;\Gamma \vdash e : \texttt{bool} \qquad \Phi;\Gamma \vdash s_b : \bot \qquad \text{TS-Skip}\ \dfrac{}{\Phi;\Gamma \vdash \texttt{skip} : \bot} \qquad \Phi;\Gamma \vdash s_r : t}{\Phi;\Gamma \vdash \underbrace{\texttt{if}\ (e)\ s_b\ \texttt{else skip}\ s_r}_{s'} : t}$$

Case: $\text{EE-Var}\ \dfrac{x : \nu_t \in \sigma}{\Phi \vdash \sigma; x \to \sigma; \nu_t}$

We simply derive:

$$\text{TE-Val}\ \frac{}{\Phi;\Gamma \vdash \nu_t : t}$$

Case: $\text{EE-Call}\ \dfrac{t\ \ell(\overline{t\ x})\ \{\ s\ \} \in \Phi}{\Phi \vdash \sigma; \underbrace{\ell(\overline{\nu_t})}_{e} \to \underbrace{\overline{x : \nu_t}}_{\sigma'}; \underbrace{\{\ s\ \}_\sigma}_{e'}}$

By this lemma's premise and with $\overline{x : t} = \Gamma' \trianglerighteq \sigma'$ we know:

$$\text{T-Prg}\ \frac{\Phi \vdash f_1 \quad \cdots \quad \text{T-Fun}\ \dfrac{\Phi; \overbrace{\overline{x : t}}^{\Gamma'} \vdash s : t}{\Phi \vdash t\ \ell(\overline{t\ x})\ \{\ s\ \}} \quad \cdots \quad \Phi \vdash f_n}{\underbrace{\vdash f_1, \ldots, t\ \ell(\overline{t\ x})\ \{\ s\ \}, \ldots, f_n}_{\Phi}}$$

Using the induction hypothesis we derive:

$$\text{TE-Stmt}\ \frac{\Phi;\Gamma' \vdash s : t}{\Phi;\Gamma' \vdash \underbrace{\{\ s\ \}_{\sigma}}_{e'} : t}$$

Case:

$$\text{EE-Stmt}\ \frac{}{\Phi \vdash \sigma;\{\ \texttt{return}\ \nu_t;\ \}_{\sigma'} \to \sigma';\nu_t}$$

We simply derive with $\Gamma' \trianglerighteq \sigma'$:

$$\text{TE-Val}\ \frac{}{\Phi;\Gamma' \vdash \nu_t : t}$$

□

B.2. VecImp

Lemma 4.5 (VecImp: Progress)
Every VecImp *term is either a term value or can be stepped into another term. To be more precise: Let* $\vdash \Phi$, $\Gamma \trianglerighteq \sigma$, *and* $l \trianglerighteq m$.

$$\textit{If}\ \begin{array}{l}\Phi;\Gamma;l \vdash s : t\\ \Phi;\Gamma;l \vdash e : t\end{array},$$

$$\textit{then}\ \begin{array}{l}s = \texttt{skip;} \ \vee\ s = \texttt{return}\ \nu_t;\\ e = \nu_t\end{array}\ \textit{or}\ \begin{array}{l}\exists s', \sigma' : \Phi; m \vdash \sigma; s \to \sigma'; s'\\ \exists e', \sigma' : \Phi; m \vdash \sigma; e \to \sigma'; e'\end{array}.$$

Proof. By mutual induction on a derivation of $\begin{array}{l}\Phi;\Gamma;l \vdash s : t\\ \Phi;\Gamma;l \vdash e : t\end{array}$.

Case:

$$\text{TS-Skip}\ \frac{}{\Phi;\Gamma;l \vdash \texttt{skip} : \bot}$$

This is the final configuration.

Case:

$$\text{TS-Ret}\ \frac{\Phi;\Gamma;l \vdash e : \tau\ \texttt{varying}(l_e) \qquad l_r = l \sqcup l_e}{\Phi;\Gamma;l \vdash \texttt{return}\ e; : \underbrace{\tau\ \texttt{varying}(l_r)}_{t}}$$

Using the induction hypothesis we deduce:

$\texttt{return}\ \nu_\tau\ \texttt{varying}(l_e)$	This is the final configuration.
otherwise:	E-Eval applies.

Case: $\text{TS-Expr}\ \dfrac{\Phi;\Gamma;l \vdash e : t_e \qquad \_ = l \sqcup \|t_e\| \qquad \Phi;\Gamma;l \vdash s : t}{\Phi;\Gamma;l \vdash e;\ s : t}$

Using the induction hypothesis we deduce:

$\nu_t;\ s$:	ES-Expr applies.
otherwise:	E-Eval applies.

Case: $\text{TS-Decl}\ \dfrac{\Phi;\Gamma[x \mapsto t_x];l \vdash s : t \qquad \_ = l \sqcup \|t_x\|}{\Phi;\Gamma;l \vdash t_x\ x;\ s : t}$

Axiomatically, ES-Decl applies.

Case: $\text{TS-Assign}\ \dfrac{l \vdash l_x \leftarrow l_e \qquad x : \tau\ \texttt{varying}(l_x) \in \Gamma \qquad \overset{\Phi;\Gamma;l \vdash \hat{s} : t}{\Phi;\Gamma;l \vdash e : \tau\ \texttt{varying}(l_e)}}{\Phi;\Gamma;l \vdash \underbrace{x = e;\ \hat{s}}_{s} : t}$

$e = \nu_\tau\ \texttt{varying}(l_e)$:

- Since

  $$x : \tau\ \texttt{varying}(l_x) \in \Gamma \quad \text{and} \quad \Gamma \trianglerighteq \sigma$$

  we know

  $$x : \nu'_{\tau\ \texttt{varying}(l_x)} \in \sigma \quad \text{for some} \quad \nu'_{\tau\ \texttt{varying}(l_x)}\ .$$

- Since
$$l \vdash l_x \leftarrow l_e \quad \text{and} \quad l \trianglerighteq m$$
we know that
$$\text{blend}(m, \nu_{\tau\ \texttt{varying}(l_e)}, \nu'_{\tau\ \texttt{varying}(l_x)})$$
is valid.
- Thus, we derive:
$$\text{ES-Assign}\ \frac{x : \nu'_{\tau\ \texttt{varying}(l_x)} \in \sigma \qquad \text{blend}(m, \nu_{\tau\ \texttt{varying}(l_e)}, \nu'_{\tau\ \texttt{varying}(l_x)})\ \text{valid}}{\Phi; m \vdash \sigma; \underbrace{x = \nu_{\tau\ \texttt{varying}(l_e)};\ \hat{s}}_{s} \rightarrow \underbrace{\sigma[x \mapsto \text{blend}(m, \nu_{\tau\ \texttt{varying}(l_e)}, \nu'_{\tau\ \texttt{varying}(l_x)})]}_{\sigma'}; \underbrace{\hat{s}}_{s'}}$$

otherwise: E-Eval applies.

Case: $e = \nu_{\texttt{bool varying}(l_c)}$

$$\text{TS-If}\ \frac{\Phi; \Gamma; l \vdash s_r : t \qquad \Phi; \Gamma; l \vdash e : \texttt{bool varying}(l_c) \qquad l' = l \sqcup l_c \qquad \Phi; \Gamma; l' \vdash s_t : \bot \qquad \Phi; \Gamma; l' \vdash s_f : \bot}{\Phi; \Gamma; l \vdash \underbrace{\texttt{if}\ (e)\ s_t\ \texttt{else}\ s_f\ s_r}_{s} : t}$$

- Since
$$l' = l \sqcup l_c \quad \text{and} \quad l \trianglerighteq m$$
we know that
$$m \wedge \nu_{\texttt{bool varying}(l_c)}$$
is valid.

- Using the induction hypothesis we derive:

$$\text{ES-IfT}\ \frac{\Phi; m \wedge \nu_{\texttt{bool varying}(l_c)} \vdash \sigma; s_t \to \sigma'; s_t' \quad m \wedge \nu_{\texttt{bool varying}(l_c)}\ \text{valid}}{\underbrace{\Phi; m \vdash \sigma; \texttt{if } (\nu_{\texttt{bool varying}(l_c)})\ s_t\ \texttt{else}\ s_f}_{s}\ s_r \to \underbrace{\sigma'; \texttt{if } (\nu_{\texttt{bool varying}(l_c)})\ s_t'\ \texttt{else}\ s_f}_{s'}\ s_r}$$

$e = \nu_{\texttt{bool varying}(l_c)}$ $s_t = \texttt{skip}$	Similar to the previous case using E-IfF.
$e = \nu_{\texttt{bool varying}(l_c)}$ $s_t = \texttt{skip}$ $s_f = \texttt{skip}$	ES-If applies.
otherwise:	E-Eval applies.

Case:

$$\text{TS-While}\ \frac{\Phi; \Gamma; l \vdash e : \texttt{bool varying}(l_c) \quad l' = l \sqcup l_c \quad \Phi; \Gamma; l' \vdash s_b : \bot \quad \Phi; \Gamma; l \vdash s_r : t}{\Phi; \Gamma; l \vdash \texttt{while } (e)\ s_b\ s_r : t}$$

Axiomatically, ES-While applies.

Case:

$$\text{TE-Var}\ \frac{x : t \in \Gamma \qquad \_ = l \sqcup \|t\|}{\Phi; \Gamma; l \vdash x : t}$$

Since $x : t \in \Gamma$ and $\Gamma \trianglelefteq \sigma$ we know $x : \nu_t \in \sigma$ for some ν_t. Thus, EE-Var applies.

Case:

$$\text{TE-Val}\ \frac{\_ = l \sqcup \|t\|}{\Phi; \Gamma; l \vdash \nu_t : t}$$

This is the final configuration.

Case: TE-Call $\dfrac{\begin{array}{c}\texttt{simd}(l_\ell)\ t\ \ell(t_{x_1}\ x_1, \dots, t_{x_n}\ x_n)\ \{\ s\ \} \in \Phi \\ \Phi;\Gamma;l \vdash e_1 : t_1 \quad \cdots \quad \Phi;\Gamma;l \vdash e_n : t_n \\ l \vdash \|t_{x_1}\| \leftarrow \|t_{e_1}\| \quad \cdots \quad l \vdash \|t_{x_n}\| \leftarrow \|t_{e_n}\| \quad l \vdash l_\ell \leftarrow l \quad \_ = l \sqcup \|t\|\end{array}}{\Phi;\Gamma;l \vdash \ell(e_1, \dots, e_n) : t}$

Using the induction hypothesis we deduce:

$\overline{e} = \overline{\nu_t}$:	Hence, EE-Call applies.
otherwise:	E-Eval applies.

Case: TE-Stmt $\dfrac{\Phi;\Gamma;l \vdash s : t \quad \_ = l \sqcup \|t\|}{\Phi;\Gamma;l \vdash \{\ s\ \}_\sigma : t}$

Using the induction hypothesis we deduce:

$s = \texttt{return}\ \nu_t;$:	EE-Stmt applies.
otherwise:	E-Eval applies.

Case: TE-Vec $\dfrac{\Phi;\Gamma;l \vdash e_1 : \tau\ \texttt{varying}(1) \quad \cdots \quad \Phi;\Gamma;l \vdash e_n : \tau\ \texttt{varying}(1) \quad \_ = l \sqcup n}{\Phi;\Gamma;l \vdash \{e_1, \dots, e_n\} : \tau\ \texttt{varying}(n)}$

Using the induction hypothesis we deduce that either E-Eval applies or we have the final configuration

$$\{\nu^1_{\tau\ \texttt{varying}(1)}, \cdots, \nu^n_{\tau\ \texttt{varying}(n)}\} = \nu_{\tau\ \texttt{varying}(n)}\ .$$

□

Lemma 4.6 (VecImp: Preservation)
If a well-typed VecImp *term takes a step of evaluation, the resulting term is also well-typed. To be more precise: Let* $\vdash \Phi$, $\Gamma \trianglerighteq \sigma$, $\Gamma' \trianglerighteq \sigma'$, *and* $l \trianglerighteq m$.

$$\textit{If}\ \begin{array}{l}\Phi;\Gamma;l \vdash s : t \\ \Phi;\Gamma;l \vdash e : t\end{array}\ \textit{and}\ \begin{array}{l}\Phi;m \vdash \sigma;s \to \sigma';s' \\ \Phi;m \vdash \sigma;e \to \sigma';e'\end{array}\ ,\ \textit{then}\ \begin{array}{l}\Phi;\Gamma';l \vdash s' : t \\ \Phi;\Gamma';l \vdash e' : t\end{array}\ .$$

Proof. By mutual induction on a derivation of $\begin{array}{l}\Phi; m \vdash \sigma; s \to \sigma'; s' \\ \Phi; m \vdash \sigma; e \to \sigma'; e'\end{array}$. Many evaluation rules do not produce a new state. Thus, $\sigma' = \sigma$ in these cases. By Lemma 4.1 this means that $\Gamma' = \Gamma$. Many case distinctions will silently make use of this insight.

Case:

$$\text{E-Eval}\ \frac{\Phi; m \vdash \sigma; \hat{a} \to \sigma'; \hat{a}'}{\Phi; m \vdash \sigma; \underbrace{\mathcal{E}[\hat{a}]}_{a} \to \sigma'; \underbrace{\mathcal{E}[\hat{a}']}_{a'}}$$

By the induction hypothesis we know

$$\Phi; \Gamma'; l \vdash \hat{a}' : t \quad \text{where} \quad \Gamma' \backsimeq \sigma' \ .$$

Using the appropriate typing rule we find

$$\Phi; \Gamma'; l \vdash \underbrace{\mathcal{E}[\hat{a}']}_{a'} : t$$

as expected:

$$\underbrace{\mathcal{E}[\hat{a}]}_{a} = \underbrace{\texttt{return}\ \hat{a};}_{a}$$

$$\text{TS-Ret}\ \frac{\Phi; \Gamma; l \vdash \hat{a}' : \tau\ \texttt{varying}(l_a) \qquad l_r = l \sqcup l_a}{\Phi; \Gamma; l \vdash \underbrace{\texttt{return}\ \hat{a}';}_{a'} : \underbrace{\tau\ \texttt{varying}(l_r)}_{t}}$$

$$\underbrace{\mathcal{E}[\hat{a}]}_{a} = \underbrace{\hat{a};\ s}_{a}$$

$$\text{TS-Expr}\ \frac{\Phi; \Gamma'; l \vdash \hat{a}' : t_a \qquad \_ = l \sqcup \|t_a\| \qquad \Phi; \Gamma'; l \vdash s : t}{\Phi; \Gamma'; l \vdash \underbrace{\hat{a}';\ s}_{a'} : t}$$

$$\underbrace{\mathcal{E}[\hat{a}]}_{a} = \underbrace{x = \hat{a};\ s}_{a}$$

$$\text{TS-Assign}\ \frac{x : \tau\ \texttt{varying}(l_x) \in \Gamma \qquad l \vdash l_x \leftarrow l_a \qquad \Phi; \Gamma; l \vdash \hat{a}' : \tau\ \texttt{varying}(l_a) \qquad \Phi; \Gamma; l \vdash s : t}{\Phi; \Gamma; l \vdash \underbrace{x = \hat{a}';\ s}_{a'} : t}$$

$$\underbrace{\mathcal{E}[\hat{a}]}_{a} = \underbrace{\texttt{if}\ (\hat{a})\ s_t\ \texttt{else}\ s_f\ s_r}_{a}$$

$$\text{TS-If}\ \frac{\Phi; \Gamma; l \vdash \hat{a}' : \texttt{bool}\ \texttt{varying}(l_c) \qquad l' = l \sqcup l_c \qquad \Phi; \Gamma; l' \vdash s_t : \bot \qquad \Phi; \Gamma; l' \vdash s_f : \bot \qquad \Phi; \Gamma; l \vdash s_r : t}{\Phi; \Gamma; l \vdash \underbrace{\texttt{if}\ (\hat{a}')\ s_t\ \texttt{else}\ s_f\ s_r}_{a'} : t}$$

$$\underbrace{\mathcal{E}[\hat{a}]}_{a} = \underbrace{\ell(\overline{\nu_t}, \hat{a}, \overline{e})}_{a}$$

$$\text{TE-Call}\ \frac{\begin{array}{c}\texttt{simd}(l_\ell)\ t\ \ell(t_{x_1}\ x_1, \ldots, t_{x_a}\ x_a, \ldots, t_{x_n}\ x_n)\ \{\ s\ \} \in \Phi \\ \Phi; \Gamma; l \vdash \nu_1 : t_1 \quad \cdots \quad \Phi; \Gamma; l \vdash \hat{a}' : t_a \quad \cdots \quad \Phi; \Gamma; l \vdash e_n : t_n \\ l \vdash \|t_{x_1}\| \leftarrow \|t_1\| \quad \cdots \quad l \vdash \|t_{x_a}\| \leftarrow \|t_a\| \quad \cdots \quad l \vdash \|t_{x_n}\| \leftarrow \|t_n\| \\ l \vdash l_\ell \leftarrow l \qquad \_ = l \sqcup \|t\|\end{array}}{\Phi; \Gamma; l \vdash \underbrace{\ell(\overline{\nu_t}, \hat{a}', \overline{e})}_{a'} : t}$$

$$\underbrace{\mathcal{E}[\hat{a}]}_{a} = \underbrace{\{\ a\ \}_{\hat{\sigma}}}_{a} \qquad \text{TE-Stmt}\ \frac{\Phi;\Gamma;l \vdash \hat{a}' : t \qquad \_ = l \sqcup \|t\|}{\Phi;\Gamma;l \vdash \underbrace{\{\ \hat{a}'\ \}_{\sigma}}_{a'} : t}$$

$$\underbrace{\mathcal{E}[\hat{a}]}_{a} = \underbrace{\{\overline{\nu_{\tau\ \texttt{varying}(1)}}, \hat{a}, \overline{e}\}}_{a} \qquad \text{TE-Vec}\ \frac{\begin{array}{c}\Phi;\Gamma;l \vdash \nu_{\tau\ \texttt{varying}(1)} : \tau\ \texttt{varying}(1) \quad \cdots \\ \Phi;\Gamma;l \vdash \hat{a}' : \tau\ \texttt{varying}(1) \quad \cdots \\ \Phi;\Gamma;l \vdash e_n : \tau\ \texttt{varying}(1) \qquad \_ = l \sqcup n\end{array}}{\Phi;\Gamma;l \vdash \underbrace{\{\overline{\nu_{\tau\ \texttt{varying}(1)}}, \ldots, \hat{a}', \ldots, e_n\}}_{a'} : \underbrace{\tau\ \texttt{varying}(n)}_{t}}$$

Case: $\text{ES-Expr}\ \dfrac{}{\Phi; m \vdash \sigma; \nu_t;\ s \to \sigma; s}$

Straightforward from the induction hypothesis.

Case: $\text{ES-Decl}\ \dfrac{}{\Phi; m \vdash \sigma; \underbrace{t_x\ x;\ \hat{s}}_{s} \to \sigma[x \mapsto 0_{t_x}]; \underbrace{\hat{s}}_{s'}}$

We know:

$$\Gamma \trianglerighteq \sigma \quad \text{and} \quad \Gamma[x \mapsto t_x] = \Gamma' \trianglerighteq \sigma' = \sigma[x \mapsto 0_{t_x}]\ .$$

Thus, by the induction hypothesis:

$$\Phi; \Gamma' \vdash s' : t\ .$$

Case: $\text{ES-Assign}\ \dfrac{x : \nu'_{\tau\ \texttt{varying}(l_x)} \in \sigma \qquad \text{blend}(m, \nu_{\tau\ \texttt{varying}(l_e)}, \nu'_{\tau\ \texttt{varying}(l_x)})\ \text{valid}}{\begin{array}{l}\Phi; m \vdash \sigma; x = \nu_{\tau\ \texttt{varying}(l_e)};\ s \to \\ \sigma[x \mapsto \text{blend}(m, \nu_{\tau\ \texttt{varying}(l_e)}, \nu'_{\tau\ \texttt{varying}(l_x)})]; s\end{array}}$

Similar to the previous case.

Case: $\text{ES-IfT}\ \dfrac{m \wedge \nu_{\texttt{bool varying}(l_c)}\ \text{valid} \qquad \Phi; m \wedge \nu_{\texttt{bool varying}(l_c)} \vdash \sigma; s_t \to \sigma'; s'_t}{\begin{array}{l}\Phi; m \vdash \sigma; \texttt{if}\ (\nu_{\texttt{bool varying}(l_c)})\ s_t\ \texttt{else}\ s_f\ s_r \to \\ \sigma'; \texttt{if}\ (\nu_{\texttt{bool varying}(l_c)})\ s'_t\ \texttt{else}\ s_f\ s_r\end{array}}$

We know

$$l \trianglerighteq m, \quad \Gamma' \trianglerighteq \sigma', \quad l' = l \sqcup l_c, \quad \Phi;\Gamma;l' \vdash s_f : \bot \quad \text{and} \quad \Phi;\Gamma;l \vdash s_r : t\ .$$

Thus, using the induction hypothesis we derive:

$$\text{TS-If}\ \frac{\Phi;\Gamma;l \vdash \nu_{\texttt{bool varying}(l_c)} : \texttt{bool varying}(l_c) \qquad \Phi;\Gamma';l' \vdash s'_t : \bot \qquad \Phi;\Gamma;l' \vdash s_f : \bot \qquad \Phi;\Gamma;l \vdash s_r : t \qquad l' = l \sqcup l_c}{\Phi;\Gamma;l \vdash \texttt{if}\ (\nu_{\texttt{bool varying}(l_c)})\ s'_t\ \texttt{else}\ s_f\ s_r : t}$$

Case:

$$\text{ES-IfF}\ \frac{}{\Phi;m \vdash \sigma;\texttt{if}\ (\texttt{false})\ s_t\ \texttt{else}\ s_f\ s_r \to \sigma; s_f \circ s_r}$$

Similar to the previous case.

Case:

$$\text{ES-If}\ \frac{}{\Phi;m \vdash \sigma;\texttt{if}\ (\nu_{\texttt{bool varying}(l)})\ \texttt{skip}\ \texttt{else}\ \texttt{skip}\ s_r \to \sigma; s_r}$$

Straightforward from the induction hypothesis.

Case:

$$\text{ES-While}\ \frac{}{\Phi;m \vdash \sigma;\underbrace{\texttt{while}\ (e)\ s_b\ s_r}_{s} \to \sigma;\underbrace{\texttt{if}\ (e)\ \{\ s_b\ \texttt{while}\ (e)\ s_b\ \}\ s_r}_{s'}}$$

Using the induction hypothesis and having the syntactic sugar in mind (see Section 4.1.1) we derive:

$$\text{TS-If}\ \frac{\Phi;\Gamma;l \vdash e : \texttt{bool varying}(l_c) \qquad \dfrac{}{\Phi;\Gamma;l' \vdash \texttt{skip} : \bot}\ \text{TS-Skip} \qquad \Phi;\Gamma;l' \vdash s_b : \bot \qquad \Phi;\Gamma;l \vdash s_r : t \qquad l' = l \sqcup l_c}{\Phi;\Gamma;l \vdash \texttt{if}\ (e)\ s_b\ \texttt{else}\ \texttt{skip}\ s_r : t}$$

Case:

$$\text{EE-Var}\ \frac{x : \nu_t \in \sigma}{\Phi;m \vdash \sigma;x \to \sigma;\nu_t}$$

We simply derive:

$$\text{TE-Val}\ \frac{}{\Phi;\Gamma \vdash \nu_t : t}$$

Case:

$$\text{EE-Call}\ \frac{\texttt{simd}(l)\ t\ \ell(\overline{t\ x})\ \{\ s\ \} \in \Phi}{\Phi; m \vdash \sigma; \underbrace{\ell(\overline{\nu_t})}_{e} \to \underbrace{\overline{x : \nu_t}}_{\sigma'}; \underbrace{\{\ s\ \}_{\sigma}}_{e'}}$$

By this lemma's premise and with $\overline{x : t} = \Gamma' \backsimeq \sigma'$ and $l \backsimeq m$ we know:

$$\text{T-Prg}\ \frac{\Phi \vdash f_1 \quad \cdots \quad \text{T-Fun}\ \dfrac{\overbrace{\Phi; \overline{x : t}; l \vdash s : t}^{\Gamma'} \quad \_ = l \sqcup \|t_1\| \quad \cdots \quad \_ = l \sqcup \|t_n\|}{\Phi \vdash \texttt{simd}(l)\ t\ \ell(\overline{t\ x})\ \{\ s\ \}} \quad \cdots \quad \Phi \vdash f_n}{\underbrace{\vdash f_1, \ldots, \texttt{simd}(l)\ t\ \ell(\overline{t\ x})\ \{\ s\ \}, \ldots, f_n}_{\Phi}}$$

Using the induction hypothesis we derive:

$$\text{TE-Stmt}\ \frac{\Phi; \Gamma'; l \vdash s : t \quad \_ = l \sqcup \|t\|}{\Phi; \Gamma'; l \vdash \underbrace{\{\ s\ \}_{\sigma}}_{e'} : t}$$

Case:

$$\text{EE-Stmt}\ \frac{}{\Phi; m \vdash \sigma; \{\ \texttt{return}\ \nu_t;\ \}_{\sigma'} \to \sigma'; \nu_t}$$

We simply derive with $\Gamma' \backsimeq \sigma'$:

$$\text{TE-Val}\ \frac{}{\Phi; \Gamma' \vdash \nu_t : t}$$

□

B.3. λ^{cps}

Lemma 9.1 (λ^{cps}: Substitution – Typing)

$$\textit{If}\ \begin{matrix}\Gamma \vdash \hat{b} \\ \Gamma \vdash \hat{e} : \hat{t}\end{matrix}, \Gamma \vdash v : t, \textit{ and } \Gamma \vdash e : t, \textit{ then}\ \begin{matrix}\Gamma \vdash [v \mapsto e]\hat{b} \\ \Gamma \vdash [v \mapsto e]\hat{e} : \hat{t}\end{matrix}.$$

Proof. By mutual induction on a derivation of $\begin{matrix}\Gamma \vdash \hat{b} \\ \Gamma \vdash \hat{e} : \hat{t}\end{matrix}$. We assume without loss of generality that any bound variables of $\begin{matrix}\hat{b} \\ \hat{e}\end{matrix}$ are different from v and from the free variables of e.

Case:

$$\text{T-Where}\ \frac{\Gamma' \coloneqq \Gamma, \ell_1 : \mathtt{cn}(\overline{t^1}), \dots, \ell_n : \mathtt{cn}(\overline{t^n}) \qquad \Gamma' \vdash \hat{b}_0 \qquad \Gamma', \overline{x^1 : t^1} \vdash \hat{b}_1 \quad \cdots \quad \Gamma', \overline{x^n : t^n} \vdash \hat{b}_n}{\Gamma \vdash \underbrace{\hat{b}_0 \ \mathtt{where}\ \ell_1(\overline{x^1 : t^1}) : \hat{b}_1, \dots, \ell_n(\overline{x^n : t^n}) : \hat{b}_n}_{\hat{b}}}$$

Using the induction hypothesis we deduce:

$$\text{T-Where}\ \frac{\Gamma' \coloneqq \Gamma, \ell_1 : \mathtt{cn}(\overline{t^1}), \dots, \ell_n : \mathtt{cn}(\overline{t^n}) \qquad \Gamma' \vdash [v \mapsto e]\hat{b}_0 \qquad \Gamma' \vdash \ell_1(\overline{x^1 : t^1}) : [v \mapsto e]\hat{b}_0 \quad \cdots \quad \Gamma' \vdash \ell_n(\overline{x^n : t^n}) : [v \mapsto e]\hat{b}_n}{\Gamma \vdash \underbrace{[v \mapsto e]\hat{b}_0 \ \mathtt{where}\ \ell_1(\overline{x^1 : t^1}) : [v \mapsto e]\hat{b}_1, \dots, \ell_n(\overline{x^n : t^n}) : [v \mapsto e]\hat{b}_n}_{[v \mapsto e]\hat{b}}}$$

Case:

$$\text{T-App}\ \frac{\Gamma \vdash \hat{e}_0 : \mathtt{cn}(t_1, \dots, t_n) \qquad \Gamma \vdash \hat{e}_1 : t_1 \quad \cdots \quad \Gamma \vdash \hat{e}_n : t_n}{\Gamma \vdash \underbrace{\hat{e}_0(\hat{e}_1, \dots, \hat{e}_n)}_{\hat{b}}}$$

Using the induction hypothesis we deduce:

$$\text{T-App}\ \frac{\Gamma \vdash [v \mapsto e]\hat{e}_0 : \mathtt{cn}(t_1, \dots, t_n) \qquad \Gamma \vdash [v \mapsto e]\hat{e}_1 : t_1 \quad \cdots \quad \Gamma \vdash [v \mapsto e]\hat{e}_n : t_n}{\Gamma \vdash \underbrace{[v \mapsto e]\hat{e}_0([v \mapsto e]\hat{e}_1, \dots, [v \mapsto e]\hat{e}_n)}_{[v \mapsto e]\hat{b}}}$$

Case:

$$\text{T-Abs}\ \frac{\ell : \mathtt{cn}(\overline{t}) \in \Gamma}{\Gamma \vdash \underbrace{\ell}_{\hat{e}} : \mathtt{cn}(\overline{t})}$$

If $v = \ell$, we have by premise:

$$\Gamma \vdash \underbrace{e}_{[v \mapsto e]\hat{e}} : \mathtt{cn}(\overline{t})$$

Otherwise we have $\hat{e} = [v \mapsto e]\hat{e}$.

Case:

$$\text{T-Param}\ \frac{x : \hat{t} \in \Gamma}{\Gamma \vdash \underbrace{x}_{\hat{e}} : \hat{t}}$$

If $v = x$, we have by premise:

$$\Gamma \vdash \underbrace{e}_{[v \mapsto e]\hat{e}} : \hat{t}$$

Otherwise we have $\hat{e} = [v \mapsto e]\hat{e}$.

Case:

$$\text{T-Clos}\ \frac{\Gamma, \overline{x : t} \vdash \hat{b}}{\Gamma \vdash \underbrace{(\ell(\overline{x : t}) : \hat{b})}_{\hat{e}} : \mathtt{cn}(\overline{t})}$$

Using the induction hypothesis we deduce:

$$\text{T-Clos}\ \frac{\Gamma, \overline{x : t} \vdash [v \mapsto e]\hat{b}}{\Gamma \vdash \underbrace{(\ell(\overline{x : t}) : [v \mapsto e]\hat{b})}_{[v \mapsto e]\hat{e}} : \mathtt{cn}(\overline{t})}$$

Case T-Primop straightforward by the induction hypothesis. □

Lemma 9.2 (λ^{cps}: →-Preservation)
If a well-typed λ^{cps} term reduces with →, the resulting term is also well-typed. To be more precise:

$$\textit{If}\ \begin{array}{c}\Gamma \vdash b\\ \Gamma \vdash e : t\end{array}\ \textit{and}\ \begin{array}{c}b \to b'\\ e \to e'\end{array}\ ,\ \textit{then}\ \begin{array}{c}\Gamma \vdash b'\\ \Gamma \vdash e' : t\end{array}\ .$$

Proof. By mutual induction on a derivation of $\begin{array}{c}b \to b'\\ e \to e'\end{array}$ and Lemma 9.1.

Case:

$$\text{R-Cong}_{1a}\ \frac{b_0 \to b_0'}{\underbrace{b_0\ \mathtt{where}\ \overbrace{\ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}^{\overline{f}}}_{b} \to \underbrace{b_0'\ \mathtt{where}\ \overline{f}}_{b'}}$$

Using the induction hypothesis we derive:

$$\text{T-Where}\ \frac{\begin{array}{c}\Gamma' \coloneqq \Gamma, \ell_1 : \mathtt{cn}(\overline{t^1}), \ldots, \ell_n : \mathtt{cn}(\overline{t^n})\\ \Gamma' \vdash b_0' \qquad \Gamma' \vdash \ell_1(\overline{x^1 : t^1}) : b_0' \quad \cdots \quad \Gamma' \vdash \ell_n(\overline{x^n : t^n}) : b_n\end{array}}{\Gamma \vdash \underbrace{b_0'\ \mathtt{where}\ \overbrace{\ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}^{\overline{f}}}_{b'}}$$

Case:

$$\text{R-Cong}_{1b}\ \frac{b_i \to b_i' \qquad 1 \le i \le m}{\underbrace{b_0\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_i(\overline{x^i : t^i}) : b_i, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}_{b} \to \underbrace{b_0\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_i(\overline{x^i : t^i}) : b_i', \ldots, \ell_n(\overline{x^n : t^n}) : b_n}_{b'}}$$

Using the induction hypothesis we derive:

$$\text{T-Where}\ \frac{\Gamma' \coloneqq \Gamma, \ell_1 : \texttt{cn}(\overline{t^1}), \ldots, \ell_n : \texttt{cn}(\overline{t^n}) \qquad \Gamma' \vdash b_0 \quad \Gamma' \vdash \ell_1(\overline{x^1 : t^1}) : b_0 \quad \cdots \quad \Gamma' \vdash \ell_i(\overline{x^i : t^i}) : b_i' \quad \cdots \quad \Gamma' \vdash \ell_n(\overline{x^n : t^n}) : b_n}{\Gamma \vdash \underbrace{b_0\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_i(\overline{x^i : t^i}) : b_i', \ldots, \ell_n(\overline{x^n : t^n}) : b_n}_{b'}}$$

Case:

$$\text{R-Cong}_{2a}\ \frac{e_0 \to e_0'}{b\ =\ e_0(\overline{e}) \to e_0'(\overline{e})\ =\ b'}$$

Using the induction hypothesis we derive:

$$\text{T-App}\ \frac{\Gamma \vdash e_0' : \texttt{cn}(t_1, \ldots, t_n) \qquad \Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash e_0'(e_1, \ldots, e_n)\ =\ b'}$$

Case:

$$\text{R-Cong}_{2b}\ \frac{e_i \to e_i'}{b\ =\ e_0(e_1, \ldots, e_i, \ldots, e_n) \to e_0(e_1, \ldots, e_i', \ldots, e_n)\ =\ b'}$$

Using the induction hypothesis we derive:

$$\text{T-App}\ \frac{\Gamma \vdash e_0 : \texttt{cn}(t_1, \ldots, t_n) \qquad \Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_i' : t_i \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash e_0(e_1, \ldots, e_i', \ldots, e_n)\ =\ b'}$$

Case:

$$\text{R-Clos}\ \frac{b \to b'}{e\ =\ (\ell(\overline{x : t}) : b) \to (\ell(\overline{x : t}) : b')\ =\ e'}$$

Using the induction hypothesis we derive:

$$\text{T-Clos}\ \frac{\Gamma, \overline{x : t} \vdash b'}{\Gamma \vdash \underbrace{(\ell(\overline{x : t}) : b')}_{e'} : \texttt{cn}(\overline{t})}$$

Cases R-Where and R-App by Lemma 9.1.
Case R-Primop straightforward by the induction hypothesis.
Case R-Fold is trivial. □

Lemma 9.5 ($\rightarrow^* = \twoheadrightarrow^*$)

(a) Whenever $\begin{matrix} b \rightarrow b' \\ e \rightarrow e' \end{matrix}$ *, then* $\begin{matrix} b \twoheadrightarrow b' \\ e \twoheadrightarrow e' \end{matrix}$.

(b) Whenever $\begin{matrix} b \twoheadrightarrow b' \\ e \twoheadrightarrow e' \end{matrix}$ *, then* $\begin{matrix} b \rightarrow^* b' \\ e \rightarrow^* e' \end{matrix}$.

(c) $\rightarrow^*$ *is the reflexive transitive closure of* $\twoheadrightarrow$.

Proof.

(a) By mutual induction on a derivation of $\begin{matrix} b \rightarrow b' \\ e \rightarrow e' \end{matrix}$ and Lemma 9.4.

Case:
$$\text{R-Cong}_{1a}\ \frac{b_0 \rightarrow b_0'}{\overline{f}}\ \underbrace{b_0\ \texttt{where}\ \overbrace{\ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}^{\overline{f}}}_{b} \rightarrow \underbrace{b_0'\ \texttt{where}\ \overline{f}}_{b'}$$

Using the induction hypothesis and Lemma 9.4 we derive:

$$\text{P-Cong}_1\ \frac{b_0 \twoheadrightarrow b_0' \quad b_1 \twoheadrightarrow b_1 \quad \cdots \quad b_n \twoheadrightarrow b_n}{\underbrace{b_0\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}_{b}}$$
$$\twoheadrightarrow \underbrace{b_0'\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}_{b'}$$

Case: $\text{R-Cong}_{1b}\ \dfrac{b_i \to b_i' \qquad 1 \le i \le m}{\underbrace{b_0\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \dots, \ell_i(\overline{x^i : t^i}) : b_i, \dots, \ell_n(\overline{x^n : t^n}) : b_n}_{b} \to \underbrace{b_0\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \dots, \ell_i(\overline{x^i : t^i}) : b_i', \dots, \ell_n(\overline{x^n : t^n}) : b_n}_{b'}}$

Using the induction hypothesis and Lemma 9.4 we derive:

$$\text{P-Cong}_1\ \frac{b_0 \twoheadrightarrow b_0 \quad \cdots \quad b_i \twoheadrightarrow b_i' \quad \cdots \quad b_n \twoheadrightarrow b_n}{\underbrace{b_0\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \dots, \ell_i(\overline{x^i : t^i}) : b_i, \dots, \ell_n(\overline{x^n : t^n}) : b_n}_{b} \twoheadrightarrow \underbrace{b_0\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \dots, \ell_i(\overline{x^i : t^i}) : b_i', \dots, \ell_n(\overline{x^n : t^n}) : b_n}_{b'}}$$

Case: $\text{R-Cong}_{2a}\ \dfrac{e_0 \to e_0'}{b\ =\ e_0(\overline{e}) \to e_0'(\overline{e})\ =\ b'}$

Using the induction hypothesis and Lemma 9.4 we derive:

$$\text{P-Cong}_2\ \frac{e_0 \twoheadrightarrow e_0' \qquad e_1 \twoheadrightarrow e_1 \quad \cdots \quad e_n \twoheadrightarrow e_n}{b\ =\ e_0(\overline{e}) \twoheadrightarrow e_0'(\overline{e})\ =\ b'}$$

Case: $\text{R-Cong}_{2b}\ \dfrac{e_i \to e_i'}{b\ =\ e_0(e_1, \dots, e_i, \dots, e_n) \to e_0(e_1, \dots, e_i', \dots, e_n)\ =\ b'}$

Using the induction hypothesis and Lemma 9.4 we derive:

$$\text{P-Cong}_2\ \frac{e_0 \twoheadrightarrow e_0 \quad \cdots \quad e_i \twoheadrightarrow e_i' \quad \cdots \quad e_n \twoheadrightarrow e_n}{b\ =\ e_0(e_1, \dots, e_i, \dots, e_n) \twoheadrightarrow e_0(e_1, \dots, e_i', \dots, e_n)\ =\ b'}$$

Case: R-Where
$$\underbrace{b_0 \texttt{ where } \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}_{b} \to \underbrace{[\ell_1 \mapsto \ell_1(\overline{x^1 : t^1}) : b_1 \texttt{ where } \overline{f}] \cdots [\ell_n \mapsto \ell_n(\overline{x^n : t^n}) : b_n \texttt{ where } \overline{f}] b_0}_{b'}$$

Using the induction hypothesis and Lemma 9.4 we derive:

$$\text{P-Where}\ \frac{b_0 \twoheadrightarrow b_0 \quad \cdots \quad b_n \twoheadrightarrow b_n}{\underbrace{b_0 \texttt{ where } \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}_{b} \twoheadrightarrow \underbrace{[\ell_1 \mapsto \ell_1(\overline{x^1 : t^1}) : b_1 \texttt{ where } \overline{f}] \cdots [\ell_n \mapsto \ell_n(\overline{x^n : t^n}) : b_n \texttt{ where } \overline{f}] b_0}_{b'}}$$

Case: R-App $\underbrace{(\ell(x_1 : t_1, \ldots, x_n : t_n) : \hat{b})(e_1, \ldots, e_n)}_{b} \to \underbrace{[x_1 \mapsto e_1] \cdots [x_n \mapsto e_n]\hat{b}}_{b'}$

Using the induction hypothesis and Lemma 9.4 we derive:

$$\text{P-App}\ \frac{\hat{b} \twoheadrightarrow \hat{b} \qquad e_1 \twoheadrightarrow e_1 \quad \cdots \quad e_n \twoheadrightarrow e_n}{\underbrace{(\ell(x_1 : t_1, \ldots, x_n : t_n) : \hat{b})(e_1, \ldots, e_n)}_{b} \twoheadrightarrow \underbrace{[x_1 \mapsto e_1] \cdots [x_n \mapsto e_n]\hat{b}}_{b'}}$$

Cases R-Clos and R-Fold are trivial.
Case R-Primop straightforward by the induction hypothesis and Lemma 9.4.

(b) By mutual induction on a derivation of $\begin{matrix} b \twoheadrightarrow b' \\ e \twoheadrightarrow e' \end{matrix}$ and transitivity of $\to^*$.

Case:
$$\text{P-Cong}_1\ \frac{b_0 \twoheadrightarrow b'_0 \quad \cdots \quad b_n \twoheadrightarrow b'_n}{\underbrace{b_0 \texttt{ where } \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}_{b} \twoheadrightarrow \underbrace{b'_0 \texttt{ where } \ell_1(\overline{x^1 : t^1}) : b'_1, \ldots, \ell_n(\overline{x^n : t^n}) : b'_n}_{b'}}$$

Using the induction hypothesis we apply R-Cong$_{1a}$ and n-times R-

Cong_{1b}. We combine the results via transitivity of $\twoheadrightarrow^*$ to obtain b'.

Case: $\text{P-Cong}_2 \dfrac{e_0 \twoheadrightarrow e_0' \quad \cdots \quad e_n \twoheadrightarrow e_n'}{e_0(e_1, \ldots, e_n) \twoheadrightarrow e_0'(e_1', \ldots, e_n')}$

Using the induction hypothesis we apply R-Cong_{2a} and n-times R-Cong_{2b}. We combine the results via transitivity of $\twoheadrightarrow^*$ to obtain b'.

Case: P-Where $\dfrac{b_0 \twoheadrightarrow b_0' \quad \cdots \quad b_n \twoheadrightarrow b_n'}{\underbrace{b_0 \texttt{ where } \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}_{b} \twoheadrightarrow \underbrace{\cdots[\ell_i \mapsto \ell_i(\overline{x^i : t^i}) : b_i' \texttt{ where } \ell_1(\overline{x^1 : t^1}) : b_1' \cdots \ell_n(\overline{x^n : t^n}) : b_n'] \cdots b_0'}_{b'}}$

Using the induction hypothesis we apply R-Cong_{1a}, n-times R-Cong_{1b} and finally R-Where. We combine the results via transitivity of $\twoheadrightarrow^*$ to obtain b'.

Case: P-App $\dfrac{b \twoheadrightarrow b' \qquad e_1 \twoheadrightarrow e_1' \quad \cdots \quad e_n \twoheadrightarrow e_n'}{\underbrace{(\ell(x_1 : t_1, \ldots, x_n : t_n) : b)(e_1, \ldots, e_n)}_{\hat{b}} \twoheadrightarrow \underbrace{[x_1 \mapsto e_1'] \cdots [x_n \mapsto e_n']b'}_{\hat{b}'}}$

Using the induction hypothesis we apply R-Cong_{2a}, n-times R-Cong_{2b} and finally R-App. We combine the results via transitivity of $\twoheadrightarrow^*$ to obtain b'.
Cases P-Abs, P-Param, and P-Fold are trivial.
Cases P-Primop and P-Clos straightforward by the induction hypothesis and transitivity of $\twoheadrightarrow^*$.

(c)
- By (a) we have $\to \subseteq \twoheadrightarrow$, hence $\to^* \subseteq \twoheadrightarrow^*$.
- By (b) we have $\twoheadrightarrow \subseteq \to^*$, hence $\twoheadrightarrow^* \subseteq \to^*$.
- Thus, $\to^* = \twoheadrightarrow^*$.

□

Lemma 9.6 (λ^{cps}: Substitution – Reduction)

$$\textit{If } \begin{matrix}\hat{b} \twoheadrightarrow \hat{b}' \\ \hat{e} \twoheadrightarrow \hat{e}'\end{matrix} \textit{ and } e \twoheadrightarrow e' \textit{ then } [v \mapsto e] \begin{matrix}\hat{b} \\ \hat{e}\end{matrix} \twoheadrightarrow [v \mapsto e'] \begin{matrix}\hat{b}' \\ \hat{e}'\end{matrix} .$$

Proof. By mutual induction on a derivation of $\begin{matrix}\hat{b} \twoheadrightarrow \hat{b}' \\ \hat{e} \twoheadrightarrow \hat{e}'\end{matrix}$. We assume without loss of generality that any bound variables of $\begin{matrix}\hat{b} \\ \hat{e}\end{matrix}$ are different from v and from the free variables of e.

Case:

$$\text{P-Cong}_1 \ \frac{b_0 \twoheadrightarrow b_0' \quad \cdots \quad b_n \twoheadrightarrow b_n'}{\underbrace{b_0 \ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}_{\hat{b}} \twoheadrightarrow \underbrace{b_0' \ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1', \ldots, \ell_n(\overline{x^n : t^n}) : b_n'}_{\hat{b}'}}$$

Using the induction hypothesis we derive:

$$\text{P-Cong}_1 \ \frac{[v \mapsto e]b_0 \twoheadrightarrow [v \mapsto e']b_0' \quad \cdots \quad [v \mapsto e]b_n \twoheadrightarrow [v \mapsto e']b_n'}{\underbrace{[v \mapsto e]b_0 \ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : [v \mapsto e]b_1, \ldots, \ell_n(\overline{x^n : t^n}) : [v \mapsto e]b_n}_{[v \mapsto e]\hat{b}} \twoheadrightarrow \underbrace{[v \mapsto e']b_0' \ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : [v \mapsto e']b_1', \ldots, \ell_n(\overline{x^n : t^n}) : [v \mapsto e']b_n'}_{[{}'S]\hat{b}'}}$$

Case:

$$\text{P-Cong}_2 \ \frac{e_0 \twoheadrightarrow e_0' \quad \cdots \quad e_n \twoheadrightarrow e_n'}{e_0(e_1, \ldots, e_n) \twoheadrightarrow e_0'(e_1', \ldots, e_n')}$$

Using the induction hypothesis we derive:

$$\text{P-Cong}_2 \ \frac{[v \mapsto e]e_0 \twoheadrightarrow [v \mapsto e']e_0 \quad \cdots \quad [v \mapsto e]e_n \twoheadrightarrow [v \mapsto e']e_n'}{\underbrace{[v \mapsto e]e_0([v \mapsto e]e_1, \ldots, [v \mapsto e]e_n)}_{[v \mapsto e]\hat{b}} \twoheadrightarrow \underbrace{[v \mapsto e']e_0'([v \mapsto e']e_1', \ldots, [v \mapsto e']e_n')}_{[v \mapsto e']\hat{b}}}$$

Case: $\text{P-Where}\ \dfrac{b_0 \twoheadrightarrow b_0' \quad \cdots \quad b_n \twoheadrightarrow b_n'}{\underbrace{b_0\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}_{\hat{b}} \twoheadrightarrow \underbrace{\cdots[\ell_i \mapsto \ell_i(\overline{x^i : t^i}) : b_i'\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1' \cdots \ell_n(\overline{x^n : t^n}) : b_n'] \cdots b_0'}_{\hat{b}'}}$

Using the induction hypothesis we derive:

$$\text{P-Where}\ \frac{[v \mapsto e]b_0 \twoheadrightarrow [v \mapsto e']b_0' \quad \cdots \quad [v \mapsto e]b_n \twoheadrightarrow [v \mapsto e']b_n'}{\underbrace{[v \mapsto e]b_0\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}_{[v \mapsto e]\hat{b}} \twoheadrightarrow \underbrace{\cdots[\ell_i \mapsto \ell_i(\overline{x^i : t^i}) : [v \mapsto e']b_i'\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : [v \mapsto e']b_1' \cdots \ell_n(\overline{x^n : t^n}) : [v \mapsto e']b_n'] \cdots]b_0}_{[v \mapsto e'] \cdots [\ell_i \mapsto \ell_i(\overline{x^i : t^i}) : b_i'\ \texttt{where}\ \ell_1(\overline{x^1 : t^1}) : b_1' \cdots \ell_n(\overline{x^n : t^n}) : b_n'] \cdots b_0' = [v \mapsto e']\hat{b}'}}$$

Case: $\text{P-App}\ \dfrac{b \twoheadrightarrow b' \qquad e_1 \twoheadrightarrow e_1' \quad \cdots \quad e_n \twoheadrightarrow e_n'}{\underbrace{(\ell(x_1 : t_1, \ldots, x_n : t_n) : b)(e_1, \ldots, e_n)}_{\hat{b}} \twoheadrightarrow \underbrace{[x_1 \mapsto e_1'] \cdots [x_n \mapsto e_n']b'}_{\hat{b}'}}$

Using the induction hypothesis we derive:

$$\text{P-App}\ \frac{[v \mapsto e]b \twoheadrightarrow [v \mapsto e']b' \qquad [v \mapsto e]e_1 \twoheadrightarrow [v \mapsto e']e_1' \quad \cdots \quad [v \mapsto e]e_n \twoheadrightarrow [v \mapsto e']e_n'}{\underbrace{[v \mapsto e](\ell(x_1 : t_1, \ldots, x_n : t_n) : b)(e_1, \ldots, e_n)}_{[v \mapsto e]\hat{b}} \twoheadrightarrow \underbrace{[x_1 \mapsto [v \mapsto e']e_1'] \cdots [x_n \mapsto [v \mapsto e']e_n'][v \mapsto e']b'}_{[v \mapsto e'][x_1 \mapsto e_1'] \cdots [x_n \mapsto e_n']b' = [v \mapsto e']\hat{b}'}}$$

Case: $\text{P-Abs}\ \dfrac{}{\hat{e} = \ell \twoheadrightarrow \ell = \hat{e}'}$

If $v = \ell$, we have by premise:

$$\underbrace{[\ell \mapsto e]\ell = e}_{[v \mapsto e]\hat{e}} \twoheadrightarrow \underbrace{e' = [\ell \mapsto e']\ell}_{[v \mapsto e']\hat{e}'}$$

Otherwise we derive:

$$\text{P-Abs}\ \frac{}{[v \mapsto e]\hat{e} \;=\; \ell \twoheadrightarrow \ell \;=\; [v \mapsto e']\hat{e}'}$$

Case: $\text{P-Param}\ \dfrac{}{\hat{e} \;=\; x \twoheadrightarrow x \;=\; \hat{e}'}$

If $v = x$, we have by premise:

$$\underbrace{[x \mapsto e]x \;=\; e}_{[v \mapsto e]\hat{e}} \twoheadrightarrow \underbrace{e' \;=\; [x \mapsto e']x}_{[v \mapsto e']\hat{e}'}$$

Otherwise we derive:

$$\text{P-Param}\ \frac{}{[v \mapsto e]\hat{e} \;=\; x \twoheadrightarrow x \;=\; [v \mapsto e']\hat{e}'}$$

Case: $\text{P-Clos}\ \dfrac{b \twoheadrightarrow b'}{\underbrace{(\ell(\overline{x : t}) : b)}_{\hat{e}} \twoheadrightarrow \underbrace{(\ell(\overline{x : t}) : b')}_{\hat{e}'}}$

Using the induction hypothesis we derive:

$$\text{P-Clos}\ \frac{[v \mapsto e]b \twoheadrightarrow [v \mapsto e']b'}{\underbrace{(\ell(\overline{x : t}) : [v \mapsto e]b)}_{[v \mapsto e]\hat{e}} \twoheadrightarrow \underbrace{(\ell(\overline{x : t}) : [v \mapsto e']b')}_{[v \mapsto e']\hat{e}'}}$$

Case P-Fold is trivial.
Case P-Primop straightforward by the induction hypothesis. □

Lemma 9.7 (λ^{cps}: Maximal parallel one-step reductions)

$$\textit{Whenever}\ \begin{matrix} b \twoheadrightarrow \hat{b} \\ e \twoheadrightarrow \hat{e} \end{matrix}\ ,\ \textit{then}\ \begin{matrix} \hat{b} \twoheadrightarrow \rho[b] \\ \hat{e} \twoheadrightarrow \rho[e] \end{matrix}\ .$$

Proof. By mutual induction on a derivation of $\begin{matrix} b \twoheadrightarrow \hat{b} \\ e \twoheadrightarrow \hat{e} \end{matrix}$ and Lemma 9.6.

Case: P-Cong_1, where $|\overline{f}| \geq 1$ $\dfrac{b_0 \twoheadrightarrow \hat{b}_0 \quad \cdots \quad b_n \twoheadrightarrow \hat{b}_n}{\underbrace{\underbrace{b_0 \texttt{ where } \overbrace{\ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}^{\overline{f}}}_{b} \twoheadrightarrow \underbrace{\hat{b}_0 \texttt{ where } \ell_1(\overline{x^1 : t^1}) : \hat{b}_1, \ldots, \ell_n(\overline{x^n : t^n}) : \hat{b}_n}_{\hat{b}}}}$

Using the induction hypothesis we derive:

$$\dfrac{\text{P-Where}\ \dfrac{\hat{b}_0 \twoheadrightarrow \rho[b_0] \quad \cdots \quad \hat{b}_n \twoheadrightarrow \rho[b_n]}{\underbrace{\hat{b}_0 \texttt{ where } \ell_1(\overline{x^1 : t^1}) : \hat{b}_1, \ldots, \ell_n(\overline{x^n : t^n}) : \hat{b}_n}_{\hat{b}}}}{\twoheadrightarrow \underbrace{\cdots[\ell_i \mapsto \ell_i(\overline{x^i : t^i}) : \rho[b_i] \texttt{ where } \ell_1(\overline{x^1 : t^1}) : \rho[b_1] \cdots \ell_n(\overline{x^n : t^n}) : \rho[b_n]] \cdots \rho[b_0]}_{\rho[b]}}$$

Case: $\text{P-Cong}_1 \dfrac{b_0 \twoheadrightarrow \hat{b}_0}{b \;=\; b_0 \twoheadrightarrow \hat{b}_0 \;=\; \hat{b}}$, no `where`-bindings

Using the induction hypothesis we derive:

$$\text{P-Cong}_1 \dfrac{\hat{b}_0 \twoheadrightarrow \rho[b_0]}{\hat{b} \;=\; \hat{b}_0 \twoheadrightarrow \rho[b_0] \;=\; \rho[b]}$$

Case: $\text{P-Cong}_2 \dfrac{e_0 \twoheadrightarrow \hat{e}_0 \quad \cdots \quad e_n \twoheadrightarrow \hat{e}_n}{\underbrace{e_0(e_1, \ldots, e_n)}_{e} \twoheadrightarrow \underbrace{\hat{e}_0(\hat{e}_1, \ldots, \hat{e}_n)}_{\hat{e}}}$, where e_0 is not a closure

Using the induction hypothesis we derive:

$$\text{P-Cong}_2 \dfrac{\hat{e}_0 \twoheadrightarrow \rho[e_0] \quad \cdots \quad \hat{e}_n \twoheadrightarrow \rho[e_n]}{\underbrace{\hat{e}_0(\hat{e}_1, \ldots, \hat{e}_n)}_{\hat{e}} \twoheadrightarrow \underbrace{\rho[e_0]\,(\rho[e_1], \ldots, \rho[e_n])}_{\rho[e]}}$$

Case:
$$\text{P-Cong}_2\ \text{P-Clos}\ \frac{\dfrac{b \twoheadrightarrow \hat{b}}{(\ell(\overline{x:t}):b) \twoheadrightarrow (\ell(\overline{x:t}):\hat{b})} \qquad e_1 \twoheadrightarrow \hat{e}_1 \quad \cdots \quad e_n \twoheadrightarrow \hat{e}_n}{\underbrace{(\ell(\overline{x:t}):b)(e_1,\ldots,e_n)}_{e} \twoheadrightarrow \underbrace{(\ell(\overline{x:t}):\hat{b})(\hat{e}_1,\ldots,\hat{e}_n)}_{\hat{e}}}\ , \text{where } e_0 = (\ell(\overline{x:t}):b)$$

Using the induction hypothesis we derive:

$$\text{P-App}\ \frac{\hat{b} \twoheadrightarrow \rho[b] \qquad \hat{e}_1 \twoheadrightarrow \rho[e_1] \quad \cdots \quad \hat{e}_n \twoheadrightarrow \rho[e_n]}{\underbrace{(\ell(x_1:t_1,\ldots,x_n:t_n):\hat{b})(\hat{e}_1,\ldots,\hat{e}_n)}_{\hat{e}} \twoheadrightarrow \underbrace{[x_1 \mapsto \rho[e_1]]\cdots[x_n \mapsto \rho[e_n]]\rho[b]}_{\rho[e]}}$$

Case:
$$\text{P-Where}\ \frac{b_0 \twoheadrightarrow \hat{b}_0 \quad \cdots \quad b_n \twoheadrightarrow \hat{b}_n}{\underbrace{b_0\ \texttt{where}\ \ell_1(\overline{x^1:t^1}):b_1,\ldots,\ell_n(\overline{x^n:t^n}):b_n}_{b}}$$
$$\twoheadrightarrow \underbrace{\cdots[\ell_i \mapsto \ell_i(\overline{x^i:t^i}):\hat{b}_i\ \texttt{where}\ \ell_1(\overline{x^1:t^1}):\hat{b}_1\cdots\ell_n(\overline{x^n:t^n}):\hat{b}_n]\cdots\hat{b}_0}_{\hat{b}}$$

Using the induction hypothesis and Lemma 9.6 we deduce:

$$\underbrace{\cdots[\ell_i \mapsto \ell_i(\overline{x^i:t^i}):\hat{b}_i\ \texttt{where}\ \ell_1(\overline{x^1:t^1}):\hat{b}_1\cdots\ell_n(\overline{x^n:t^n}):\hat{b}_n]\cdots\hat{b}_0}_{\hat{b}}$$
$$\twoheadrightarrow \underbrace{\cdots[\ell_i \mapsto \ell_i(\overline{x^i:t^i}):\rho[b_i]\ \texttt{where}\ \ell_1(\overline{x^1:t^1}):\rho[b_1]\cdots\ell_n(\overline{x^n:t^n}):\rho[b_n]]\cdots\rho[b_0]}_{\rho[b]}$$

Case:
$$\text{P-App}\ \frac{b_{clos} \twoheadrightarrow \hat{b}_{clos} \qquad e_1 \twoheadrightarrow \hat{e}_1 \quad \cdots \quad e_n \twoheadrightarrow \hat{e}_n}{\underbrace{(\ell(x_1:t_1,\ldots,x_n:t_n):b_{clos})(e_1,\ldots,e_n)}_{b} \twoheadrightarrow \underbrace{[x_1 \mapsto \hat{e}_1]\cdots[x_n \mapsto \hat{e}_n]\hat{b}_{clos}}_{\hat{b}}}$$

Using the induction hypothesis and Lemma 9.6 we deduce:

$$\underbrace{[x_1 \mapsto \hat{e}_1]\cdots[x_n \mapsto \hat{e}_n]\hat{b}_{clos}}_{\hat{b}} \twoheadrightarrow \underbrace{[x_1 \mapsto \rho[e_1]]\cdots[x_n \mapsto \rho[e_n]]\hat{b}_{clos}}_{\rho[b]}$$

Case:

$$\text{P-Clos}\ \frac{b \twoheadrightarrow \hat{b}}{\underbrace{(\ell(\overline{x:t}):b)}_{e} \twoheadrightarrow \underbrace{(\ell(\overline{x:t}):\hat{b})}_{\hat{e}}}$$

Using the induction hypothesis we derive:

$$\text{P-Clos}\ \frac{\hat{b} \twoheadrightarrow \rho[b]}{\underbrace{(\ell(\overline{x:t}):\hat{b})}_{\hat{e}} \twoheadrightarrow \underbrace{(\ell(\overline{x:t}):\rho[b])}_{\rho[e]}}$$

Cases P-Abs, P-Param, and P-Fold are trivial.
Case P-Primop straightforward by the induction hypothesis. □

Lemma 10.2 (λ^{cps}: $\Rightarrow$-Progress)
Every λ^{cps} term is either a term value or can be stepped with $\Rightarrow$ into another term. To be more precise:

$$\textit{If}\ \begin{array}{c}\Gamma \vdash b\\ \Gamma \vdash e : t\end{array}\ ,\ \textit{then}\ \begin{array}{c} b = \texttt{exit}(\nu)\\ e = \nu\end{array}\ \textit{or}\ \begin{array}{c}\exists b' : b \Rightarrow b'\\ \exists e' : e \Rightarrow e'\end{array}\ .$$

Proof. By mutual induction on a derivation of $\begin{array}{c}\Gamma \vdash b\\ \Gamma \vdash e : t\end{array}$.

Case:

$$\text{T-Where}\ \frac{\Gamma' \vdash b_0 \qquad \overbrace{\begin{array}{c}\Gamma' \coloneqq \Gamma, \ell_1 : \texttt{cn}(\overline{t^1}), \ldots, \ell_n : \texttt{cn}(\overline{t^n})\\ \Gamma' \vdash \ell_1(\overline{x^1 : t^1}) : b_0 \quad \cdots \quad \Gamma' \vdash \ell_n(\overline{x^n : t^n}) : b_n\end{array}}^{\overline{f}}}{\Gamma \vdash \underbrace{b_0\ \texttt{where}\ \overbrace{\ell_1(\overline{x^1 : t^1}) : b_1, \ldots, \ell_n(\overline{x^n : t^n}) : b_n}^{\overline{f}}}_{b}}$$

Using the induction hypothesis we deduce:

$b_0\Downarrow, \ldots, b_i\Downarrow, \ldots, b_n\Downarrow$:	E-Eval applies.
$b_0\Downarrow, \ldots, b_n\Downarrow$:	E-Where applies.

Case: $$\text{T-App}\ \frac{\Gamma \vdash e_0 : \texttt{cn}(t_1, \ldots, t_n) \qquad \Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash \underbrace{e_0(e_1, \ldots, e_n)}_{b}}$$

Using the induction hypothesis we deduce:

$e_0\Downarrow, \ldots, e_i\Downarrow, \ldots, e_n\Downarrow$:	E-Eval applies.
$e_0\Downarrow, \ldots, e_n\Downarrow, e_0$ closure with corresponding arity:	E-App applies.

Note that E-Skip can never be triggered in a well-typed program. See Section 10.3.1.
Cases T-Abs, T-Param, T-Primop, and T-Clos are trivial. □

Acronyms

ANF administrative normal form

AST abstract syntax tree

AoSoA array of structures of arrays

AoS array of structures

ArBB Intel® Array Building Blocks

BTA binding-time analysis

CFA control-flow analysis

CFF control-flow form

CFG control-flow graph

CPS continuation-passing style

CPU central processing unit

DAG directed acyclic graph

DSL domain-specific language

GCC GNU Compiler Collection

GHC Glasgow Haskell Compiler

GPU graphics processing unit

ICC Intel® C/C++ compiler

IR intermediate representation

ISA instruction set architecture

JIT just-in-time

LMS lightweight modular staging

MIC Intel® Many Integrated Core Architecture

PCF Programming Computable Functions

PDG program dependence graph

PDW program dependence web

RV region vectorizer

SCC strongly connected component

SIMD single instruction, multiple data

SLP superword-level parallelism

SPMD single program, multiple data

SSA static single assignment

SoA structure of array

WFV whole-function vectorization

ISPC Intel® SPMD compiler

primop primitive operation

cogen compiler generator

gegen generating extension generator

Bibliography

[Adv+03] Vikram S. Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. "LLVA: A Low-level Virtual Instruction Set Architecture". In: *Proceedings of the 36th Annual International Symposium on Microarchitecture, San Diego, CA, USA, December 3-5, 2003*. 2003, pp. 205–216. DOI: 10.1109/MICRO.2003.1253196. (Cit. on pp. 59, 89).

[AH02] Tomas Akenine-Möller and Eric Haines. *Real-time rendering, 2nd Edition*. A K Peters, 2002. ISBN: 978-1-56881-182-6 (cit. on p. 7).

[AK01] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001. ISBN: 1-55860-286-0 (cit. on p. 15).

[AK87] Randy Allen and Ken Kennedy. "Automatic Translation of Fortran Programs to Vector Form". In: *ACM Trans. Program. Lang. Syst.* 9.4 (1987), pp. 491–542. DOI: 10.1145/29873.29875. (Cit. on p. 15).

[All+83] John R. Allen, Ken Kennedy, Carrie Porterfield, and Joe D. Warren. "Conversion of Control Dependence to Data Dependence". In: *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*. 1983, pp. 177–189. DOI: 10.1145/567067.567085. (Cit. on p. 15).

[App06] Andrew W. Appel. *Compiling with Continuations (corr. version)*. Cambridge University Press, 2006. ISBN: 978-0-521-03311-4 (cit. on pp. 88, 95, 97, 104, 113, 178, 186).

[App98] Andrew W. Appel. “SSA is Functional Programming”. In: *SIGPLAN Notices* 33.4 (1998), pp. 17–20. DOI: 10.1145/278283.278285. (Cit. on p. 101).

[Bac59] John W. Backus. “The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference”. In: *IFIP Congress*. 1959, pp. 125–131 (cit. on p. 192).

[Bar84] H.P. Barendregt. *The lambda calculus: its syntax and semantics*. Studies in logic and the foundations of mathematics. North-Holland, 1984. ISBN: 9780444867483 (cit. on pp. 96, 121).

[BBZ11] Matthias Braun, Sebastian Buchwald, and Andreas Zwinkau. *Firm—A Graph-Based Intermediate Representation*. Tech. rep. 35. Karlsruhe Institute of Technology, 2011. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000025470 (cit. on p. 104).

[BEM00] William L. Briggs, Henson Van Emden, and Stephen F. McCormick. *A multigrid tutorial (2. ed.)* SIAM, 2000. ISBN: 978-0-89871-462-3 (cit. on p. 159).

[BJ93] Anders Bondorf and Jesper Jørgensen. “Efficient Analysis for Realistic Off-Line Partial Evaluation”. In: *J. Funct. Program.* 3.3 (1993), pp. 315–346. DOI: 10.1017/S0956796800000769. (Cit. on pp. 132, 136).

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. “The Program Dependence Web: A Representation Supporting Control, Data, and Demand-Driven Interpretation of Imperative Languages”. In: *Proceedings of the ACM SIGPLAN’90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*. 1990, pp. 257–271. DOI: 10.1145/93542.93578. (Cit. on p. 102).

[Bou+92] Richard J. Boulton, Andrew D. Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. “Experience with Embedding Hardware Description Languages in HOL”. In: *Theorem Provers in Circuit Design, Proceedings of the IFIP TC10/WG 10.2 International Conference*

on Theorem Provers in Circuit Design: Theory, Practice and Experience, Nijmegen, The Netherlands, 22-24 June 1992, Proceedings. 1992, pp. 129–156 (cit. on p. 81).

[BR13] Yosi Ben-Asher and Nadav Rotem. "Hybrid type legalization for a sparse SIMD instruction set". In: *TACO* 10.3 (2013), 11:1–11:14. DOI: 10.1145/2509420.2509422. (Cit. on p. 73).

[Bra+13] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. "Simple and Efficient Construction of Static Single Assignment Form". In: *Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013, pp. 102–122. DOI: 10.1007/978-3-642-37051-9_6. (Cit. on pp. 59, 92, 108, 145).

[Bro+11] Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. "A Heterogeneous Parallel Framework for Domain-Specific Languages". In: *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*. 2011, pp. 89–100. DOI: 10.1109/PACT.2011.15. (Cit. on p. 135).

[Bud+02] Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. "Fast Copy Coalescing and Live-Range Identification". In: *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*. 2002, pp. 25–32. DOI: 10.1145/512529.512534. (Cit. on p. 113).

[BW94] Lars Birkedal and Morten Welinder. "Hand-Writing Program Generator Generators". In: *Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94, Madrid, Spain, September 14-16, 1994, Proceedings*. 1994, pp. 198–214. DOI: 10.1007/3-540-58402-1_15. (Cit. on p. 132).

[CH06] Felice Cardone and J. Roger Hindley. *History of Lambda-calculus and Combinatory Logic*. Tech. rep. MRRS-05-06. Swansea University Mathematics Department Research Report, 2006 (cit. on p. 95).

[Cha+10] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. "Language virtualization for heterogeneous parallel computing". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. 2010, pp. 835–847. DOI: 10.1145/1869459.1869527. (Cit. on pp. 82, 84, 134).

[Chi+12] Charisee Chiw, Gordon L. Kindlmann, John H. Reppy, Lamont Samuels, and Nick Seltzer. "Diderot: a parallel DSL for image analysis and visualization". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 2012, pp. 111–120. DOI: 10.1145/2254064.2254079. (Cit. on p. 20).

[Chu32] Alonzo Church. "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics* 33.2 (1932), pp. 346–366. ISSN: 0003486X. URL: http://www.jstor.org/stable/1968337 (cit. on p. 95).

[CKS07] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. "Finally Tagless, Partially Evaluated". In: *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*. 2007, pp. 222–238. DOI: 10.1007/978-3-540-76637-7_15. (Cit. on pp. 82, 85, 134, 135).

[CL11] William R. Cook and Ralf Lämmel. "Tutorial on Online Partial Evaluation". In: *Proceedings IFIP Working Conference on Domain-Specific Languages, DSL 2011, Bordeaux, France, 6-8th September 2011*. 2011, pp. 168–180. DOI: 10.4204/EPTCS.66.8. (Cit. on pp. 132, 136).

[Cli95] Clifford Noel Click Jr. “Combining Analyses, Combining Optimizations”. UMI Order No. GAX96-10626. PhD thesis. Houston, TX, USA, 1995 (cit. on p. 104).

[Coc02] W. Paul Cockshott. “Vector Pascal an array language for multimedia code”. In: *APL*. 2002, pp. 83–91. DOI: 10.1145/602231.602242. (Cit. on p. 17).

[Con93] Charles Consel. “A Tour of Schism: A Partial Evaluation System For Higher-Order Applicative Languages”. In: *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93, Copenhagen, Denmark, June 14-16, 1993*. 1993, pp. 145–154. DOI: 10.1145/154630.154645. (Cit. on p. 137).

[CP95] Cliff Click and Michael Paleczny. “A Simple Graph-Based Intermediate Representation”. In: *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*. 1995, pp. 35–49. DOI: 10.1145/202529.202534. (Cit. on p. 102).

[CR14] Julien Cretin and Didier Rémy. “System F with coercion constraints”. In: *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*. 2014, 34:1–34:10. DOI: 10.1145/2603088.2603128. (Cit. on p. 95).

[CR36] A. Church and J. B. Rosser. “Some properties of conversion”. In: *Transactions of the American Mathematical Society* 39 (1936). Electronic Edition, pp. 472–482. URL: http://www.jstor.org/stable/2268573 (cit. on p. 125).

[Cyt+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”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991), pp. 451–490. DOI: 10.1145/115372.115320. (Cit. on pp. 59, 101).

[Cza+03] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. "DSL Implementation in MetaOCaml, Template Haskell, and C++". In: *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*. 2003, pp. 51–72. DOI: 10.1007/978-3-540-25935-0_4. (Cit. on p. 133).

[Dan+14] Piotr Danilewski, Marcel Köster, Roland Leißa, Richard Membarth, and Philipp Slusallek. "Specialization through dynamic staging". In: *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*. 2014, pp. 103–112. DOI: 10.1145/2658761.2658774. (Cit. on p. 134).

[De 72] Nicolaas Govert De Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem". In: *INDAG. MATH* 34 (1972), pp. 381–392 (cit. on pp. 96, 97).

[De 80] Nicolaas Govert De Bruijn. "A survey of the project AUTOMATH". In: *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by J. P. Seldin and J. R. Hindley. Academic Press, 1980, pp. 589–606 (cit. on p. 96).

[DeV+11] Zach DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. "Liszt: a domain specific language for building portable mesh-based PDE solvers". In: *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*. 2011, 9:1–9:12. DOI: 10.1145/2063384.2063396. (Cit. on p. 135).

[DeV+13] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. "Terra: a multi-stage language for high-performance computing". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 2013, pp. 105–116. DOI: 10.1145/2462156.2462166. (Cit. on pp. 20, 134).

[DS00] Olivier Danvy and Ulrik Pagh Schultz. “Lambda-dropping: transforming recursive equations into programs with block structure”. In: *Theor. Comput. Sci.* 248.1-2 (2000), pp. 243–287. DOI: 10.1016/S0304-3975(00)00054-2. (Cit. on pp. 101, 166, 184).

[Est+14] Pierre Estérie, Joel Falcou, Mathias Gaunard, and Jean-Thierry Lapresté. “Boost.SIMD: generic programming for portable SIMDization”. In: *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing, WPMVP 2014, Orlando, Florida, USA, February 16, 2014*. 2014, pp. 1–8. DOI: 10.1145/2568058.2568063. (Cit. on p. 16).

[Fea91] Paul Feautrier. “Dataflow analysis of array and scalar references”. In: *International Journal of Parallel Programming* 20.1 (1991), pp. 23–53. DOI: 10.1007/BF01407931. (Cit. on p. 16).

[Fea92a] Paul Feautrier. “Some efficient solutions to the affine scheduling problem. I. One-dimensional time”. In: *International Journal of Parallel Programming* 21.5 (1992), pp. 313–347. DOI: 10.1007/BF01407835. (Cit. on p. 16).

[Fea92b] Paul Feautrier. “Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time”. In: *International Journal of Parallel Programming* 21.6 (1992), pp. 389–420. DOI: 10.1007/BF01379404. (Cit. on p. 16).

[FI73] Adin D. Falkoff and Kenneth E. Iverson. “The Design of APL”. In: *IBM Journal of Research and Development* 17.5 (1973), pp. 324–334. DOI: 10.1147/rd.174.0324. (Cit. on p. 17).

[Fla+93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. “The Essence of Compiling with Continuations”. In: *Proceedings of the ACM SIGPLAN’93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*. 1993, pp. 237–247. DOI: 10.1145/155090.155113. (Cit. on p. 100).

[FLS04] Nicolas Fritz, Philipp Lucas, and Philipp Slusallek. "CGiS, a new Language for Data-parallel GPU Programming". In: *Proceedings of the Vision, Modeling, and Visualization Conference 2004 (VMV 2004), Stanford, California, USA, November 16-18, 2004*. 2004, pp. 241–248 (cit. on p. 19).

[Fly72] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Trans. Computers* 21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071. (Cit. on p. 3).

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (1987), pp. 319–349. DOI: 10.1145/24039.24041. (Cit. on p. 102).

[Fum+17] Juan José Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. "Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation". In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*. 2017, pp. 60–73. DOI: 10.1145/3050748.3050761. (Cit. on p. 133).

[Fut99] Yoshihiko Futamura. "Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler". In: *Higher-Order and Symbolic Computation* 12.4 (Dec. 1999). Revision of the 1971 paper, pp. 381–391. ISSN: 1573-0557. DOI: 10.1023/A:1010095604496. (Cit. on pp. 132, 136).

[GF10] Carlos H. Gonzalez and Basilio B. Fraguela. "A Generic Algorithm Template for Divide-and-Conquer in Multicore Systems". In: *12th IEEE International Conference on High Performance Computing and Communications, HPCC 2010, 1-3 September 2010, Melbourne, Australia*. 2010, pp. 79–88. DOI: 10.1109/HPCC.2010.24. (Cit. on p. 183).

[Ghu+07] Anwar Ghuloum, Terry Smith, Gansha Wu, Xin Zhou, Jesse Fang, Peng Guo, Byoungro So, Mohan Rajagopalan, Yongjian Chen, and Biao Chen. "Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture". In: *Intel Technology Journal* 11.04 (Nov. 2007) (cit. on p. 17).

[Glü12] Robert Glück. “A self-applicable online partial evaluator for recursive flowchart languages”. In: *Softw., Pract. Exper.* 42.6 (2012), pp. 649–673. DOI: 10.1002/spe.1086. (Cit. on p. 132).

[GW14] Jeremy Gibbons and Nicolas Wu. “Folding domain-specific languages: deep and shallow embeddings (functional Pearl)”. In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. 2014, pp. 339–347. DOI: 10.1145/2628136.2628138. (Cit. on p. 81).

[Haf15] Immanuel Haffner. “Sierra: A SIMD Extension for C++”. Advisors: Sebastian Hack and Roland Leißa. Bachelor’s thesis. Saarland University, Apr. 1, 2015 (cit. on p. 13).

[Hai+16] Michael Haidl, Michel Steuwer, Tim Humernbrum, and Sergei Gorlatch. “Multi-stage programming for GPUs in C++ using PACXX”. In: *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit, GPGPU@PPoPP 2016, Barcelona, Spain, March 12 - 16, 2016*. 2016, pp. 32–41. DOI: 10.1145/2884045.2884049. (Cit. on p. 134).

[Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977. ISBN: 0444002057 (cit. on p. 183).

[HG14] Michael Haidl and Sergei Gorlatch. “PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14”. In: *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM 2014, New Orleans, LA, USA, November 17, 2014*. 2014, pp. 1–11. DOI: 10.1109/LLVM-HPC.2014.9. (Cit. on p. 134).

[HL90] Pat Hanrahan and Jim Lawson. “A language for shading and lighting calculations”. In: *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1990, Dallas, TX, USA, August 6-10, 1990*. 1990, pp. 289–298. DOI: 10.1145/97879.97911. (Cit. on p. 19).

[Hof+08] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. "Polymorphic embedding of dsls". In: *Generative Programming and Component Engineering, 7th International Conference, GPCE 2008, Nashville, TN, USA, October 19-23, 2008, Proceedings*. 2008, pp. 137–148. DOI: 10.1145/1449913.1449935. (Cit. on p. 134).

[HR99] Michael R. Hansen and Hans Rischel. *Introduction to Programming Using Sml*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201398206 (cit. on p. 171).

[HU72] Matthew S. Hecht and Jeffrey D. Ullman. "Flow Graph Reducibility". In: *SIAM J. Comput.* 1.2 (1972), pp. 188–202. DOI: 10.1137/0201014. (Cit. on p. 176).

[Hud98] Paul Hudak. "Modular domain specific languages and tools". In: *Proceedings of the Fifth International Conference on Software Reuse, ICSR 1998, Victoria, BC, Canada, June 2-5, 1998*. 1998, pp. 134–142. DOI: 10.1109/ICSR.1998.685738. (Cit. on pp. 134, 135).

[Hum+14] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. "A domain-specific language for building self-optimizing AST interpreters". In: *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*. 2014, pp. 123–132. DOI: 10.1145/2658761.2658776. (Cit. on p. 133).

[Int13] Intel Corporation. *Intel® Cilk™ Plus Language Extension Specification*. Version 1.2. 2013 (cit. on p. 18).

[Int16] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-033. June 2016 (cit. on p. 3).

[ISO11] ISO. *ISO/IEC 14882:2011(E). Information technology – Programming languages – C++*. Geneva, Switzerland, 2011 (cit. on pp. 23, 27).

[JG02] Neil D. Jones and Arne J. Glenstrup. “Program generation, termination, and binding-time analysis”. In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.* 2002, p. 283. DOI: 10.1145/581478.581505. (Cit. on p. 137).

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice Hall international series in computer science. Prentice Hall, 1993. ISBN: 978-0-13-020249-9 (cit. on p. 137).

[Joh85] Thomas Johnsson. “Lambda Lifting: Treansforming Programs to Recursive Equations”. In: *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings.* 1985, pp. 190–203. DOI: 10.1007/3-540-15975-4_37. (Cit. on pp. 140, 166, 184).

[Jon95] Neil D. Jones. “Special Address: MIX ten years after”. In: *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, USA, June 21-23, 1995.* 1995, pp. 24–38. DOI: 10.1145/215465.215468. (Cit. on p. 132).

[Jør98] Jesper Jørgensen. “SIMILIX: A Self-Applicable Partial Evaluator for Scheme”. In: *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998.* 1998, pp. 83–107. DOI: 10.1007/3-540-47018-2_3. (Cit. on p. 137).

[Jov+14] Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. “Yin-yang: concealing the deep embedding of DSLs”. In: *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014.* 2014, pp. 73–82. DOI: 10.1145/2658761.2658771. (Cit. on pp. 84, 135).

[JS86] Ulrik Jørring and William L. Scherlis. “Compilers and Staging Transformations”. In: *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986.*

1986, pp. 86–96. DOI: 10.1145/512644.512652. (Cit. on p. 132).

[JSS89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. "Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation". In: *Lisp and Symbolic Computation* 2.1 (1989), pp. 9–50 (cit. on p. 132).

[Kar15] Ralf Karrenberg. *Automatic SIMD Vectorization of SSA-based Control Flow Graphs*. Springer, 2015. ISBN: 978-3-658-10112-1. DOI: 10.1007/978-3-658-10113-8. (Cit. on p. 20).

[Kel95] Richard Kelsey. "A Correspondence between Continuation Passing Style and Static Single Assignment Form". In: *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*. 1995, pp. 13–23. DOI: 10.1145/202529.202532. (Cit. on pp. 101, 177, 180, 181).

[Ken07] Andrew Kennedy. "Compiling with continuations, continued". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*. 2007, pp. 177–190. DOI: 10.1145/1291151.1291179. (Cit. on p. 101).

[KH11] Ralf Karrenberg and Sebastian Hack. "Whole-function vectorization". In: *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. 2011, pp. 141–150. DOI: 10.1109/CGO.2011.5764682. (Cit. on p. 20).

[KH12] Ralf Karrenberg and Sebastian Hack. "Improving Performance of OpenCL on CPUs". In: *Compiler Construction - 21st International Conference, CC 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, pp. 1–20. DOI: 10.1007/978-3-642-28652-0_1. (Cit. on p. 20).

[Khr12] Khronos OpenCL Working Group. *The OpenCL Specification*. Version: 1.2. 2012 (cit. on pp. 19, 152).

[Khr13] Khronos OpenCL Working Group. *The OpenGL Shading Language*. Language Version: 4.40. 2013 (cit. on p. 19).

[Khr14] Khronos OpenCL Working Group. *The SPIR™ Specification – Standard Portable Intermediate Representation*. Version 1.2. 2014 (cit. on p. 152).

[Khr15] Khronos OpenCL Working Group. *SYCL™ Specification*. Version 1.2. 2015 (cit. on pp. 6, 85, 135).

[KKS13] Ralf Karrenberg, Marek Kosta, and Thomas Sturm. "Presburger Arithmetic in Memory Access Optimization for Data-Parallel Languages". In: *Frontiers of Combining Systems - 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings*. 2013, pp. 56–70. DOI: 10.1007/978-3-642-40885-4_5. (Cit. on p. 20).

[Kle38] Stephen Cole Kleene. "On Notation for Ordinal Numbers". In: *J. Symb. Log.* 3.4 (1938), pp. 150–155. DOI: 10.2307/2267778. (Cit. on pp. 130, 140).

[Klo+14] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. "Building Efficient Query Engines in a High-Level Language". In: *PVLDB* 7.10 (2014), pp. 853–864. URL: http://www.vldb.org/pvldb/vol7/p853-klonatos.pdf (cit. on p. 82).

[Kös+14] Marcel Köster, Roland Leißa, Sebastian Hack, Richard Membarth, and Philipp Slusallek. "Code Refinement of Stencil Codes". In: *Parallel Processing Letters* 24.3 (2014). DOI: 10.1142/S0129626414410035. (Cit. on p. 93).

[Kra+86] David A. Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. "Orbit: an optimizing compiler for scheme (with retrospective)". In: *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*. 1986, pp. 175–191. DOI: 10.1145/989393.989414. (Cit. on pp. 97, 178).

[Kum15] Suhas Kumar. *Fundamental Limits to Moore's Law*. 2015. eprint: arXiv:1511.05956 (cit. on p. vii).

[KW92] Morry Katz and Daniel Weise. "Towards a New Perspective on Partial Evaluation". In: *PEPM'92, ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Fairmont Hotel, San Francisco, CA, USA, June 19-20, 1992, Proceedings (TR YALEU/DCS/RR-909)*. 1992, pp. 29–37 (cit. on p. 136).

[LA00] Samuel Larsen and Saman P. Amarasinghe. "Exploiting superword level parallelism with multimedia instruction sets". In: *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*. 2000, pp. 145–156. DOI: 10.1145/349299.349320. (Cit. on p. 16).

[Lei+15] Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. "Shallow embedding of DSLs via online partial evaluation". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2015, Pittsburgh, PA, USA, October 26-27, 2015*. Best Paper Award. 2015, pp. 11–20. DOI: 10.1145/2814204.2814208. (Cit. on pp. 93, 135, 150).

[LHH14] Roland Leißa, Immanuel Haffner, and Sebastian Hack. "Sierra: a SIMD extension for C++". In: *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing, WPMVP 2014, Orlando, Florida, USA, February 16, 2014*. 2014, pp. 17–24. DOI: 10.1145/2568058.2568062. (Cit. on p. 13).

[LHW12] Roland Leißa, Sebastian Hack, and Ingo Wald. "Extending a C-like language for portable SIMD programming". In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*. 2012, pp. 65–74. DOI: 10.1145/2145816.2145825. (Cit. on pp. 12, 19, 67).

[LKH15] Roland Leißa, Marcel Köster, and Sebastian Hack. "A graph-based higher-order intermediate representation". In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*. 2nd place: Arti-

fact Evaluation for CGO/PPoPP'15. 2015, pp. 202–212. DOI: 10.1109/CGO.2015.7054200. (Cit. on p. 93).

[Lom67] Lionello Lombardi. "Incremental Computation: The Preliminary Design of a Programming System Which Allows for Incremental Data Assimilation in Open-Ended Man-Computer Information Systems". In: *Advances in Computers* 8 (1967), pp. 247–333. DOI: 10.1016/S0065-2458(08)60698-1. (Cit. on p. 132).

[Mar+03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. "Cg: a system for programming graphics hardware in a C-like language". In: *ACM Trans. Graph.* 22.3 (2003), pp. 896–907. DOI: 10.1145/882262.882362. (Cit. on p. 19).

[Mem+14a] Richard Membarth, Oliver Reiche, Christian Schmitt, Frank Hannig, Jürgen Teich, Markus Stürmer, and Harald Köstler. "Towards a performance-portable description of geometric multigrid algorithms using a domain-specific language". In: *J. Parallel Distrib. Comput.* 74.12 (2014), pp. 3191–3201. DOI: 10.1016/j.jpdc.2014.08.008. (Cit. on p. 162).

[Mem+14b] Richard Membarth, Philipp Slusallek, Marcel Köster, Roland Leißa, and Sebastian Hack. "Target-specific refinement of multigrid codes". In: *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC '14, New Orleans, Louisiana, USA, November 16-21, 2014*. 2014, pp. 52–57. DOI: 10.1109/WOLFHPC.2014.5. (Cit. on p. 93).

[Mem+16] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. "HIPA^{cc}: A Domain-Specific Language and Compiler for Image Processing". In: *IEEE Trans. Parallel Distrib. Syst.* 27.1 (2016), pp. 210–224. DOI: 10.1109/TPDS.2015.2394802. (Cit. on pp. 85, 135, 162).

[Mer03] Jason Merrill. "Generic and gimple: A new tree representation for entire functions". In: *In Proceedings of the 2003 GCC Summit*. 2003 (cit. on pp. 59, 89).

[Mid12] Jan Midtgaard. "Control-flow analysis of functional programs". In: *ACM Comput. Surv.* 44.3 (2012), 10:1–10:33. DOI: 10.1145/2187671.2187672. (Cit. on p. 146).

[MMH13] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml - Functional Programming for the Masses*. O'Reilly, 2013. ISBN: 978-1-4493-2391-2. URL: http://shop.oreilly.com/product/0636920024743.do#tab_04_2 (cit. on p. 95).

[Moo00] Gordon E. Moore. "Readings in Computer Architecture". In: ed. by Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. Chap. Cramming More Components Onto Integrated Circuits, pp. 56–59. ISBN: 1-55860-539-8. URL: http://dl.acm.org/citation.cfm?id=333067.333074 (cit. on p. vii).

[MS06] Matthew Might and Olin Shivers. "Environment analysis via Delta CFA". In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. 2006, pp. 127–140. DOI: 10.1145/1111037.1111049. (Cit. on p. 185).

[MVB15] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. "PolyMage: Automatic Optimization for Image Processing Pipelines". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. 2015, pp. 429–443. DOI: 10.1145/2694344.2694364. (Cit. on p. 20).

[Mye13] Andrew Myers. "CS 6110: Advanced Programming Languages. Lecture 5. IMP: Big-Step and Small-Step Semantics". 2013 (cit. on p. 39).

[New+11] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael D. McCool, Anwar M. Ghuloum, Stefanus Du Toit, Zhi-Gang Wang, Zhaohui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. "Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language". In: *Proceedings of the CGO 2011, The 9th International Sym-*

posium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011. 2011, pp. 224–235. DOI: 10.1109/CGO.2011.5764690. (Cit. on pp. 17, 135).

[Ngo95] Viet Nhu Ngo. "Parallel Loop Transformation Techniques for Vector-based Multiprocessor Systems". UMI Order No. GAX94-33091. PhD thesis. Minneapolis, MN, USA, 1995 (cit. on p. 15).

[NH06] Dorit Nuzman and Richard Henderson. "Multi-platform Auto-vectorization". In: *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006), 26-29 March 2006, New York, New York, USA*. 2006, pp. 281–294. DOI: 10.1109/CGO.2006.25. (Cit. on p. 16).

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications - a formal introduction*. Wiley professional computing. Wiley, 1992. ISBN: 978-0-471-92980-2 (cit. on p. 146).

[Nor98] M. Norrish. *C Formalised in HOL*. Technical report. University of Cambridge. Computer Laboratory. University of Cambridge, Computer Laboratory, 1998. URL: https://books.google.ba/books?id=iEQiAQAAIAAJ (cit. on p. 39).

[NRZ06] Dorit Nuzman, Ira Rosen, and Ayal Zaks. "Auto-vectorization of interleaved data for SIMD". In: *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*. 2006, pp. 132–143. DOI: 10.1145/1133981.1133997. (Cit. on p. 16).

[Nuz+11] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. "Vapor SIMD: Auto-vectorize once, run everywhere". In: *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. 2011, pp. 151–160. DOI: 10.1109/CGO.2011.5764683. (Cit. on p. 16).

[NVI14] NVIDIA Corporation. *NVVM IR Specification 1.3. Reference Guide*. 2014 (cit. on p. 152).

[NVI17] NVIDIA Corporation. *CUDA C Programming Guide. Design Guide*. 2017 (cit. on pp. 19, 152).

[NZ08] Dorit Nuzman and Ayal Zaks. “Outer-loop vectorization: revisited for short SIMD architectures”. In: *17th International Conference on Parallel Architecture and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*. 2008, pp. 2–11. DOI: 10.1145/1454115.1454119. (Cit. on p. 15).

[Ofe+13] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. “Spiral in scala: towards the systematic construction of generators for performance libraries”. In: *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*. 2013, pp. 125–134. DOI: 10.1145/2517208.2517228. (Cit. on p. 135).

[Ope13] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. 2013 (cit. on pp. 6, 18).

[Ope15] OpenACC-Standard.org. *The OpenACC® Application Programming Interface*. Version 2.5. 2015 (cit. on p. 6).

[Pér+17] Arsène Pérard-Gayot, Martin Weier, Richard Membarth, Philipp Slusallek, Roland Leißa, and Sebastian Hack. “RaTrace: simple and efficient abstractions for BVH ray traversal algorithms”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017*. 2017, pp. 157–168. DOI: 10.1145/3136040.3136044. (Cit. on p. 187).

[Pfe92] Frank Pfenning. *A Proof of the Church-Rosser Theorem and Its Representation in a Logical Framework*. Tech. rep. Pittsburgh, PA, USA: Carnegie Mellon University, 1992 (cit. on p. 124).

[PJ15] Vasileios Porpodas and Timothy M. Jones. “Throttling Automatic Vectorization: When Less is More”. In: *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*. 2015, pp. 432–444. DOI: 10.1109/PACT.2015.32. (Cit. on p. 16).

[Plo77] Gordon D. Plotkin. “LCF Considered as a Programming Language”. In: *Theor. Comput. Sci.* 5.3 (1977), pp. 223–255. DOI: 10.1016/0304-3975(77)90044-5. (Cit. on pp. 89, 113).

[PM02] Simon Peyton Jones and Simon Marlow. “Secrets of the Glasgow Haskell Compiler Inliner”. In: *J. Funct. Program.* 12.5 (July 2002), pp. 393–434. ISSN: 0956-7968. DOI: 10.1017/S0956796802004331. (Cit. on p. 96).

[PM12] Matt Pharr and William R. Mark. “ispc: A SPMD Compiler for High-Performance CPU Programming”. In: *Innovative Parallel Computing (InPar)*. 2012. DOI: 10.1109/InPar.2012.6339601 (cit. on p. 19).

[PMJ15] Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. “PSLP: padded SLP automatic vectorization”. In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*. 2015, pp. 190–201. DOI: 10.1109/CGO.2015.7054199. (Cit. on p. 16).

[Pol95] Robert Pollack. *Polishing Up the Tait-Martin-Löf Proof of the Church-Rosser Theorem*. 1995 (cit. on pp. 121, 122).

[Pro+13] Sander Pronk, Szilárd Páll, Roland Schulz, Per Larsson, Pär Bjelkmar, Rossen Apostolov, Michael R. Shirts, Jeremy C. Smith, Peter M. Kasson, David van der Spoel, Berk Hess, and Erik Lindahl. “GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit”. In: *Bioinformatics* 29.7 (2013), pp. 845–854. DOI: 10.1093/bioinformatics/btt055. (Cit. on p. 134).

[PS94] Jens Palsberg and Michael I. Schwartzbach. “Binding-time Analysis: Abstract Interpretation versus Type Inference”. In: *Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages, May 16-19, 1994, Toulouse, France*. 1994, pp. 277–288. DOI: 10.1109/ICCL.1994.288372. (Cit. on p. 135).

[PVC01] Michael Paleczny, Christopher A. Vick, and Cliff Click. “The Java HotSpot Server Compiler”. In: *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*. 2001. URL: http:

//www.usenix.org/publications/library/proceedings/jvm01/paleczny.html (cit. on p. 104).

[Rag+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 2013, pp. 519–530. DOI: 10.1145/2462156.2462176. (Cit. on pp. 82, 135).

[Rat+17] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. McRae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. "Firedrake: Automating the Finite Element Method by Composing Abstractions". In: *ACM Trans. Math. Softw.* 43.3 (2017), 24:1–24:27. DOI: 10.1145/2998441. (Cit. on pp. 82, 134).

[Rei+17] Oliver Reiche, Christof Kobylko, Frank Hannig, and Jürgen Teich. "Auto-vectorization for image processing DSLs". In: *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2017, Barcelona, Spain, June 21-22, 2017*. 2017, pp. 21–30. DOI: 10.1145/3078633.3081039. (Cit. on p. 20).

[RO10] Tiark Rompf and Martin Odersky. "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs". In: *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*. 2010, pp. 127–136. DOI: 10.1145/1868294.1868314. (Cit. on p. 134).

[Rom+14] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. "Surgical precision JIT compilers". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edin-*

burgh, United Kingdom - June 09 - 11, 2014. 2014, pp. 41–52. DOI: 10.1145/2594291.2594316. (Cit. on p. 135).

[Ruf93] Erik Steven Ruf. "Topics in Online Partial Evaluation". UMI Order No. GAX93-26550. PhD thesis. Stanford, CA, USA, 1993 (cit. on p. 132).

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Global Value Numbers and Redundant Computations". In: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*. 1988, pp. 12–27. DOI: 10.1145/73560.73562. (Cit. on p. 101).

[San95] André L. M. de Santos. "Compilation by Transformation in Non-Strict Functional Languages". PhD thesis. University of Glasgow, 1995 (cit. on pp. 100, 105, 184).

[SC11] Amin Shali and William R. Cook. "Hybrid partial evaluation". In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. 2011, pp. 375–390. DOI: 10.1145/2048066.2048098. (Cit. on pp. 132, 136).

[Sel08] Peter Selinger. "Lecture notes on the lambda calculus". In: *CoRR* abs/0804.3434 (2008). arXiv: 0804.3434. URL: http://arxiv.org/abs/0804.3434 (cit. on p. 121).

[SHC05] Jaewook Shin, Mary W. Hall, and Jacqueline Chame. "Superword-Level Parallelism in the Presence of Control Flow". In: *3nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005), 20-23 March 2005, San Jose, CA, USA*. 2005, pp. 165–175. DOI: 10.1109/CGO.2005.33. (Cit. on p. 16).

[Shi+08] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer III. "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization". In: *Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008, Island of Kos, Greece, June 7-12, 2008*. 2008, pp. 277–288. DOI: 10.1145/1375527.1375568. (Cit. on p. 19).

[Shi91] Olin Grigsby Shivers. “Control-flow Analysis of Higher-order Languages of Taming Lambda”. UMI Order No. GAX91-26964. PhD thesis. Pittsburgh, PA, USA, 1991 (cit. on pp. 146, 185).

[Smo15] Gert Smolka. “Confluence and Normalization in Reduction Systems. Lecture Notes”. Semantics lecture, winter semester 2015/2016 by Derek Dreyer and Gert Smolka. Dec. 16, 2015. URL: https://www.ps.uni-saarland.de/courses/sem-ws15/ars.pdf (cit. on p. 121).

[SRD17] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. “Lift: a functional data-parallel IR for high-performance GPU code generation”. In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. 2017, pp. 74–85. URL: http://dl.acm.org/citation.cfm?id=3049841 (cit. on p. 96).

[SSH15] Sigurd Schneider, Gert Smolka, and Sebastian Hack. “A Linear First-Order Functional Intermediate Language for Verified Compilers”. In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. 2015, pp. 344–358. DOI: 10.1007/978-3-319-22102-1_23. (Cit. on p. 113).

[Sta+16] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. “Optimizing R language execution via aggressive speculation”. In: *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*. 2016, pp. 84–95. DOI: 10.1145/2989225.2989236. (Cit. on p. 133).

[Ste78] Guy L. Steele Jr. *Rabbit: A Compiler for Scheme*. Tech. rep. Cambridge, MA, USA: Massachusetts Institute of Technology, 1978 (cit. on pp. 97, 178).

[Ste95] Bjarne Steensgaard. “Sparse Functional Stores for Imperative Programs”. In: *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR’95), San Francisco, CA, USA, January 22, 1995*. 1995, pp. 62–70. DOI: 10.1145/202529.202536. (Cit. on p. 120).

[Str00] Christopher Strachey. “Fundamental Concepts in Programming Languages”. In: *Higher-Order and Symbolic Computation* 13.1/2 (2000), pp. 11–49. DOI: 10.1023/A:1010000313106. (Cit. on p. 120).

[Str07] Bjarne Stroustrup. “Evolving a language in and for the real world: C++ 1991-2006”. In: *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*. 2007, pp. 1–59. DOI: 10.1145/1238844.1238848. (Cit. on p. 77).

[Suj+11] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. “OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning”. In: *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*. 2011, pp. 609–616 (cit. on p. 134).

[SW05] S. St-Laurent and E. Wolfgang. *The Complete HLSL Reference*. Paradoxal Press, 2005 (cit. on p. 19).

[Tai67] William W. Tait. “Intensional Interpretations of Functionals of Finite Type I”. In: *J. Symb. Log.* 32.2 (1967), pp. 198–212. DOI: 10.2307/2271658. (Cit. on p. 144).

[Tak95] M. Takahashi. “Parallel Reductions in λ-Calculus”. In: *Inf. Comput.* 118.1 (Apr. 1995), pp. 120–127. ISSN: 0890-5401. DOI: 10.1006/inco.1995.1057. (Cit. on p. 124).

[TC10] Andrew Tolmach and Tim Chevalier. *An External Representation for the GHC Core Language*. Tech. rep. The GHC Team, 2010 (cit. on p. 95).

[Tob+11] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. “Languages as libraries”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 2011, pp. 132–141. DOI: 10.1145/1993498.1993514. (Cit. on p. 134).

[Tri+09] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. "Polyhedral-Model Guided Loop-Nest Auto-Vectorization". In: *PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 12-16 September 2009, Raleigh, North Carolina, USA*. 2009, pp. 327–337. DOI: 10.1109/PACT.2009.18. (Cit. on p. 16).

[TS00] Walid Taha and Tim Sheard. "MetaML and multi-stage programming with explicit annotations". In: *Theor. Comput. Sci.* 248.1-2 (2000), pp. 211–242. DOI: 10.1016/S0304-3975(00)00053-0. (Cit. on p. 134).

[TS01] Ulrich Trottenberg and Anton Schuller. *Multigrid*. Orlando, FL, USA: Academic Press, Inc., 2001. ISBN: 0-12-701070-X (cit. on p. 159).

[Vel98] Todd L. Veldhuizen. "C++ Templates as Partial Evaluation". In: *CoRR* cs.PL/9810010 (1998). URL: http://arxiv.org/abs/cs.PL/9810010 (cit. on p. 134).

[Wal+01] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. "Interactive Rendering with Coherent Ray Tracing". In: *Comput. Graph. Forum* 20.3 (2001), pp. 153–165. DOI: 10.1111/1467-8659.00508. (Cit. on p. 3).

[Wal+17] Ingo Wald, Gregory P. Johnson, J. Amstutz, Carson Brownlee, Aaron Knoll, J. Jeffers, J. Gunther, and Paul A. Navrátil. "OSPRay - A CPU Ray Tracing Framework for Scientific Visualization". In: *IEEE Trans. Vis. Comput. Graph.* 23.1 (2017), pp. 931–940. DOI: 10.1109/TVCG.2016.2599041. (Cit. on p. 37).

[Wal12] Ingo Wald. "Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture". In: *IEEE Trans. Vis. Comput. Graph.* 18.1 (2012), pp. 47–57. DOI: 10.1109/TVCG.2010.251. (Cit. on p. 71).

[Wan+14] Haichuan Wang, Peng Wu, Ilie Gabriel Tanase, Mauricio J. Serrano, and José E. Moreira. "Simple, portable and fast SIMD intrinsic programming: generic simd library". In: *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing, WPMVP 2014, Orlando, Florida,*

USA, February 16, 2014. 2014, pp. 9–16. DOI: 10.1145/2568058.2568059. (Cit. on p. 16).

[Wim+17] Christian Wimmer, Vojin Jovanovic, Erik Eckstein, and Thomas Würthinger. "One compiler: deoptimization to optimized code". In: *Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017*. 2017, pp. 55–64. URL: http://dl.acm.org/citation.cfm?id=3033025 (cit. on p. 133).

[Wür+13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. "One VM to rule them all". In: *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. 2013, pp. 187–204. DOI: 10.1145/2509578.2509581. (Cit. on p. 133).

[Wür+17] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. "Practical partial evaluation for high-performance dynamic language runtimes". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 2017, pp. 662–676. DOI: 10.1145/3062341.3062381. (Cit. on p. 133).

[Wür14] Thomas Würthinger. "Graal and truffle: modularity and separation of concerns as cornerstones for building a multipurpose runtime". In: *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*. 2014, pp. 3–4. DOI: 10.1145/2584469.2584663. (Cit. on p. 133).

[ZR02] Jingren Zhou and Kenneth A. Ross. "Implementing database operations using SIMD instructions". In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*. 2002, pp. 145–156. DOI: 10.1145/564691.564709. (Cit. on p. 3).