Formal Semantics for the Halide Language

Alex Reinking
Gilbert Bernstein
Jonathan Ragan-Kelley

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-40
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-40.html

May 1, 2020
FORMAL SEMANTICS FOR THE HALIDE LANGUAGE

BY ALEX REINKING

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Jonathan Ragan-Kelley
Research Advisor

Date

Professor Alvin Cheung
Second Reader

Date
Formal Semantics for the Halide Language

Alex Reinking Gilbert Bernstein Jonathan Ragan-Kelley

December 20, 2019

Abstract

Halide is a domain-specific language for building and optimizing numerical pipelines for applications in image processing, neural networks, and linear algebra. Its distinctive separation of what to compute — the algorithm — from the organization of that computation — the schedule — enables programmers to focus on the optimization of their programs without reasoning about loop index edge cases, concurrency issues, or portability. Halide then ensures memory safety through a bounds inference engine that determines safe sizes for every buffer in the generated code.

In this paper, we present the Halide language in a formal setting, provide semantics for every part and proofs of correctness for the scheduling directives, and define the bounds inference problem in terms of program synthesis. We believe this precision will bolster future work in extending Halide to support a broader computational model, assist validating the implementation against a formal specification, and inspire the design of new languages and tools that apply programmer-controlled scheduling to other domains.

1 Introduction

Halide is a domain-specific language for building and optimizing dense array processing pipelines for applications like image processing, machine learning, and linear algebra [30]. Its design separates the specification of what is to be computed, known as the algorithm, from the specification of the order in which those computations should be carried out and placed in memory, known as the schedule, both of which are supplied by the user. Halide popularized this design, which is now being explored in a new generation of languages and compilers [10, 38, 29, 43, 2, 36, 25, 39, 3]. Halide is also being used widely in industry, from YouTube, to every Android phone, to Adobe Photoshop [31, 21, 33].

This design makes a fundamental safety and correctness promise: the algorithm serves as a ground truth for what the overall program is meant to compute, so the scheduling directives must not alter the outputs for any set of inputs. The key power of these languages is that programmers are thus relieved from troubleshooting large classes of bugs that arise when optimizing loops for memory locality, vectorization, or multi-core parallelism because these transformations are available in the scheduling language. This enables a schedule-centric workflow where the majority of effort is spent exploring different optimizations, not ensuring correctness after each attempted one. The safety and correctness of languages with user-controlled scheduling, however, has never been formally defined or analyzed. This paper presents the first formal definition and analysis of the core of Halide.

Formalizing Halide presents several challenges. First, it is not obvious how to formulate a metatheory: what should be true about such a language, and how should it be formally structured? Second, this formalization is complicated by trying to retain fidelity to the practical system. As an industrial system required to replace hand-optimized assembly, on esoteric accelerator architectures, in performance-critical production applications, Halide is unsurprisingly more complex than many academically-motivated array languages.

An unusual characteristic of the Halide language is its combination of lazy and eager semantics. The algorithm language is fundamentally lazy, defining a dataflow graph on infinite arrays. The core job of schedules is to prescribe particular eager, imperative implementations of these lazy, functional algorithms: a scheduled program yields a specific loop nest and buffer allocation that computes subarrays of each stage to produce the desired region of the output. Schedules include classic loop transformations, but their most
unique constructs (**compute-at**, **store-at**) specify the granularity of producer-consumer communication in a lazy computation graph. Then, to relieve the burden on the programmer of determining safe and correct extents for all of the resulting loops and intermediate buffer allocations given a schedule, Halide relies on automatic **bounds inference**. The complex interaction of all of these features has made changing the language and compiler very difficult in practice, which is a core motivation for our formalization.

This paper makes the following contributions:

- We give the first complete semantics and meta-theory for user-specified scheduling of a high performance array processing language.
- We give the first precise description of the core of the practical Halide system: the algorithm language, scheduling operators, and bounds inference problem.
- We define Halide’s bounds inference problem in terms of program synthesis.
- We prove that Halide programs are memory safe.
- We prove that scheduling does not change the output of a Halide program, provided bounds assertions pass.
- We find and fix bugs in, and make design improvements to, the practical Halide system.

2 An Example Halide Program

To introduce key concepts and build intuition for the formalism to follow, let’s start with an example. Consider a separable two-stage blur algorithm (a **pipeline**), as shown in Figure 1 (a). Blurring algorithms work by taking a weighted average of the values in a window around each pixel. A separable blur can be factored into horizontal and vertical **stages**. For this example, we make the blur window size \( w \) a **pipeline parameter**. Despite the apparent simplicity of this algorithm, a straightforward C implementation is many times slower than even a moderately optimized version [30].

The algorithm uses three **funcs**, introduced by **fun**: \( f \) modeling the input image, \( g \) the image after horizontal blurring, and \( h \) the image after vertical blurring. Unlike most array languages, Halide treats these **funcs** as partial functions defined on an unbounded \( n \)-dimensional integer lattice, rather than simply arrays. The last **func** \( h \) defines the pipeline output, which—by prohibiting recursion—is defined by acyclic dependence on other **funcs**. For simplicity of formalization, we omit treatment of special input **funcs**, using procedural inputs such as the sinusoid pattern \( \sin(x + y) \). The practical system supports pipeline input images.\(^1\)

**Funcs** may be defined as pure functions (e.g., \( f \)) or via imperative updates (e.g., \( g \) and \( h \)). Regardless, all **funcs** must begin with a **pure definition** to ensure they are defined over the entire domain. \( f \) is defined as a 2D sinusoid, while \( g \) and \( h \) are initialized to 0. After this, a **func** may optionally include one or more **update stages** that modify this initial definition. These update stages may use **reduction domains** (rdoms), which may be read as for-loops over a \((\text{base}, \text{extent})\) interval. \( g \) and \( h \) use this feature to implement a sum (a reduction) over the blur window, which is the interval \((-w, 2w+1)\). Finally, to evaluate a pipeline \( P \), we must supply parameters \( w \) and a **realization** window, e.g. \(((0, 640), (0, 480))\). Every pixel in this range of the output is then computed.

Infinite data-structures like **funcs** can be handled with lazy, functional evaluation, but then how are eager, imperative rdoms integrated? In Section 4, we show how to resolve this issue with a lazy, big-step reference semantics for the algorithm language. These semantics have no knowledge of memory, bounds, or computation order.

In order to recover an eager, imperative implementation, we **lower** the pipeline into a second, imperative, target language with C-like semantics. While there exist sensible choices for loop iteration bounds and buffer sizes, our blur algorithm never specified these. Therefore, this initial lowered program leaves symbolic **holes** in the code (written ‘?’). Figure 1 (c) shows the lowered pipeline.

\(^1\)The bounds of the images must be checked for consistency with the downstream program. This check happens as soon as the program starts running.
pipeline $P(w)$

\[
\text{fun } f(x, y) = \sin(x + y)
\]

\[
\text{fun } g(x, y) = \{ 0; \}
\]

\[
\text{rdom}(r = (-w, 2w + 1)) \quad \text{in} \quad (x, y) \leftarrow g(x, y) + f(x + r, y); \}
\]

\[
\text{fun } h(x, y) = \{ 0; \}
\]

\[
\text{rdom}(r = (-w, 2w + 1)) \quad \text{in} \quad (x, y) \leftarrow h(x, y) + g(x, y + r); \}
\]

(a) Two-stage blur algorithm with parameterized box size.

pipeline $P(w)((b_x, l_x), (b_y, l_y))$

allocate ($f(??, ??)$)

label $f$ :

label $s_1$ : for $y$ in $(??, ??)$ do for $x$ in $(??, ??)$ do $f[x, y] \leftarrow \sin(x + y)$

allocate ($g(??, ??)$)

label $g$ :

label $s_1$ : for $y$ in $(??, ??)$ do for $x$ in $(??, ??)$ do $g[x, y] \leftarrow 0$

label $s_1$ : for $y$ in $(??, ??)$ do for $x$ in $(??, ??)$ do $g[x, y] \leftarrow g[x, y] + f[x + r, y]$

allocate ($h(??, ??)$)

label $h$ :

label $s_1$ : for $y$ in $(??, ??)$ do for $x$ in $(??, ??)$ do $g[x, y] \leftarrow 0$

label $s_1$ : for $y$ in $(??, ??)$ do for $x$ in $(??, ??)$ do $h[x, y] \leftarrow h[x, y] + g[x, y + r]$

(b) Two-stage blur schedule.

pipeline $P(w)((b_x, l_x), (b_y, l_y))$

allocate ($f(??, ??)$)

for $y$ in $(??, ??)$ do for $x$ in $(??, ??)$ do $f[x, y] \leftarrow \sin(x + y)$

allocate ($g(??, ??)$)

for $y$ in $(??, ??)$ do for $x$ in $(??, ??)$ do $g[x, y] \leftarrow 0$

allocate ($h(??, ??)$)

for $y$ in $(??, ??)$ do for $x$ in $(??, ??)$ do $h[x, y] \leftarrow 0$

allocate ($s(??, ??)$)

for $y$ in $(??, ??)$ do for $x$ in $(??, ??)$ do $s[x, y] \leftarrow s[x, y] + f[x + r, y]$

allocate ($l(??, ??)$)

for $y$ in $(??, ??)$ do for $x$ in $(??, ??)$ do $l[x, y] \leftarrow l[x, y] + g[x, y + r]$

(c) Initial lowering of two-stage blur.

(d) Final scheduled two-stage blur with inferred bounds.

Figure 1: Top: Example algorithm and schedule. Bottom: Lowered target before and after scheduling and bounds inference.

\[
\text{let } h = b_y + 32 y_0 + y, \text{ in } \]

\[
\text{if } y < b_y + l_y \text{ in } \]

\[
\text{let } x = b_x + 256 x_0 + x, \text{ in } \]

\[
\text{if } x < b_x + l_x \text{ in } \]

\[
\text{let } h[x, y] \leftarrow h[x, y] + g[x, y + r] \]

\[
\text{for } y_0 \text{ in } (0, ?_{23}/32) \text{ do for } y_i \text{ in } (0, 32) \text{ do } \]

\[
\text{for } x_0 \text{ in } (0, 32/256) \text{ do for } x_i \text{ in } (0, 256) \text{ do } \]

\[
\text{for } r \text{ in } (w, 2w + 1) \text{ do } \]

\[
\text{let } x = ?_{24} + 256 x_0 + x_0 \text{ in } \]

\[
\text{let } y = ?_{22} + 32 y_0 + y, \text{ in } \]

\[
\text{if } y < ?_{22} + ?_{23} \text{ then } \]

\[
\text{if } x < ?_{24} + ?_{25} \text{ then } \]

\[
\text{h}[x, y] \leftarrow h[x, y] + g[x, y + r] \]

\[
\text{for } r \text{ in } (-w, 2w + 1) \text{ do } \]

\[
\text{let } y = b_y + 32 y_0 + y, \text{ in } \]

\[
\text{if } y < b_y + l_y \text{ in } \]

\[
\text{let } x = b_x + 256 x_0 + x, \text{ in } \]

\[
\text{if } x < b_x + l_x \text{ in } \]

\[
\text{h}[x, y] \leftarrow h[x, y] + g[x, y + r] \]

\[
\text{for } r \text{ in }uristic bounds inference.

Figure 2: h’s update stage partway through scheduling.
To fill these holes, Halide uses a process called bounds inference, which we formalize as a program synthesis problem. For our purposes, we assume a bounds inference oracle returns satisfying expressions to fill every hole, without any particular guarantee on the tightness of these results (it is always safe to allocate and compute more than is necessary). While the practical system also provides no hard guarantees, it works well enough to produce high quality code for industrial use on broad classes of programs.

The result of bounds inference is an executable program in the target language, with guaranteed memory safety and correctness with respect to the original algorithm.

However, the initial lowering is not especially efficient. Every stage of every func is computed in its own loop nest, with all rdoms translated to innermost loops. This default schedule is only one such valid schedule, albeit with poor locality and excessive memory use.

A faster schedule is specified in Figure 1 (b) using a programmer-visible language of scheduling directives. This schedule breaks the computation of \( h \) into tiles of size \( 256 \times 32 \) (\texttt{split, split, swap}). Each column of tiles is computed in parallel, independently of every other column (\texttt{traverse}). Rather than compute and store all of \( g \) before proceeding to \( h \), this schedule computes \( g \) per-tile (compute-\texttt{at}, store-\texttt{at}). At the cost of a small amount of redundant recomputation, this schedule increases locality dramatically.

In this formalization, each scheduling directive may be viewed as an individual, primitive program transformation, through which correctness and safety are preserved. For instance, Figure 2 shows the program after both \texttt{split} transformations, but before \texttt{swap}. As such, each modified program with holes can be completed via bounds inference into a hole-free target program.

Halide ensures safety by constraining the space of valid program transformations. For instance, only loops tied to pure variables (e.g. \( x, y \)) rather than reduction variables (e.g. \( r \)) may be \texttt{split, swapped}, or \texttt{traversed} in parallel.

With the blur pipeline, we can clearly see the benefits of decoupled scheduling. In a traditional high performance language like C, a programmer would need to write code similar to the generated code in Figure 1 (d). By instead factoring these scheduling strategies into a small language of directives, Halide programmers are able to explore the space of safe, equivalent programs. The formalism presented here explains how and why these guarantees are achieved.

## 3 Overview

### 3.1 Formal system structure

Halide programs are defined by an algorithm and schedule pair, each written in two related languages, denoted \( P \in \text{Alg} \) and \( T \in \text{Sched} \). The syntax and semantics for \text{Alg} (§4) are not meant to suggest a practical implementation, but instead serve as a reference meaning for an unscheduled program.

Before execution, a program \( P \) is lowered to a program in an imperative intermediate language, \( \text{Tgt} \) (§5). This language has a variant with labeled holes allowed in place of expressions, denoted \( \text{Tgt}^2 \). The function for lowering \( P \) to such a program is denoted \( \mathcal{L} : \text{Alg} \rightarrow \text{Tgt}^2 \) (§6).

If an empty schedule is provided, then \( S_0 = \mathcal{L}(P) \) is passed to a bounds inference oracle (§7) which returns some program \( \text{Tgt}' \subseteq \text{Tgt} \). The programs produced by the oracle are identical in structure to the original, but have had their holes filled by expressions that will allow the program to run and produce correct results without memory errors.

If a non-empty schedule with directives \( s_1; \ldots; s_n \) is provided, we define a corresponding sequence of programs (§8) with holes \( S_i \in \text{Tgt}^2 \) by \( S_0 = \mathcal{L}(P), S_i = S(s_i, S_{i-1}) \). Here, the function \( \mathcal{S} : \text{Sched} \times \text{Tgt}^2 \rightarrow \text{Tgt} \) applies a scheduling directive \( s_i \) to a program \( S_i \) and returns the result. Each of these programs may be further elaborated to a hole-free program \( \text{Tgt}' \subseteq \text{Tgt} \). Some final program \( \text{Tgt}^n \) is the overall result of compilation. Figure 3 illustrates the high level structure of this formal system.

### 3.2 Metatheory

Before getting into the details of the formalism, we will review our main theoretical goals. Halide makes two fundamental promises to programmers: memory safety and equivalence under scheduling transformations.

\[^2\text{The practical system conservatively infers bounds with interval arithmetic.}\]
Definition 3.1 (Input). Let \( P \) be a pipeline with \( m \) parameters and an \( n \) dimensional output func. Then an input \( z \) to \( P \) is an assignment to those \( m \) parameters and an assignment of \( n \) constant intervals defining a realization window \( R(z) \).

Programs produced by bounds inference are allowed to compute a larger window than was requested by the input. This is possible, for instance, when the output func contains an RDom that is not contained by the realization. Two pipelines are considered equivalent as long as they agree on the requested window.

Definition 3.2 (Output equivalence). Let each of \( P \) and \( P' \) be either an algorithm or target language program, with identical output dimension and parameters. They are said to have equivalent outputs for input \( z \) if for every point \( x \in R(z) \), the partial functions \( f \) returned by \( P(z) \) and \( f' \) returned by \( P'(z) \) have \( f(x) = f'(x) \). When this is the case, we write \( P \equiv_z P' \).

Definition 3.3 (Algorithm confluence). Let \( P \in \text{Alg} \) be some Halide algorithm and let \( S \in \text{Tgt} \) be some target language program with holes. We say that \( S \) is confluent with \( P \) if for all \( P' \in \text{BI}(S) \) and all inputs \( z \), either \( P(z) \) contains an error value in \( R(z) \), \( P'(z) \) contains an error in \( R(z) \), \( P'(z) \) fails an assertion check, or \( P' \equiv_z P \).

Error values and assertion failures are detailed in Sections 4.1 and 5.2 respectively. We are now able to state the two fundamental theorems about Halide. In stating these theorems, we assume that algorithm language programs are valid (§4.3), as are schedules (§8).

Theorem 3.4 (Memory safety). Let \( P \in \text{Alg} \) be a valid program, \( z \) input parameters to \( P \), and \( T \) a valid schedule for the program \( P \). Then, for all target language programs \( P' \in \text{BI}(S(T, L(P))) \), \( P'(z) \) will not access any out of bounds memory.

Theorem 3.5 (Scheduling equivalence). Let \( P \in \text{Alg} \) be a valid program, \( z \) input parameters to \( P \), and \( T \) a valid schedule for the program \( P \). Then all target language programs \( P' \in \text{BI}(S(T, L(P))) \) are confluent with \( P \).

Memory safety will be guaranteed by the bounds inference oracle. The problem posed to this oracle is defined in Section 7 such that this property is correct by construction.

4 Algorithm Language

4.1 Expressions

Figure 4 shows the abstract syntax of the expression language. The set of values \( V \) in the formal language extends \( \mathbb{Z} \), on which the usual primitive expressions are defined\(^3\):

\[
V = \mathbb{Z} \cup \{ \varepsilon_{\text{arith}}, \varepsilon_{\text{rdom}}, \varepsilon_{\text{mem}} \}
\]

Definition 4.1 (Error value). The special expression values \( \varepsilon_{\text{arith}} < \varepsilon_{\text{rdom}} < \varepsilon_{\text{mem}} \) encode a hierarchy of errors. Any operation in the expression language involving one or more of these values evaluates to the greatest among them.

\(^3\)The practical system supports floating-point and fixed-width integers, and faces standard semantic issues with those.
\[ c ::= i \in \mathbb{V} \quad \text{constants} \]
\[ a ::= f[e_1, \ldots, e_n] \quad \text{func access} \]
\[ e ::= c \mid a \mid v \mid \text{op}(e_1, \ldots, e_k) \quad \text{expression} \]
\[ I ::= (e_{\min}, e_{\len}) \quad \text{interval} \]
\[ v ::= x \quad \text{pure variable} \]
\[ \mid r \quad \text{reduction variable} \]
\[ \mid p \quad \text{parameter variable} \]
\[ \text{op} ::= + \mid - \mid \times \mid \text{div} \mid \text{mod} \quad \text{arithmetic} \]
\[ \mid \lor \mid \land \mid \neg \mid < \mid > \mid = \quad \text{logical} \]
\[ \mid \text{select} \mid \text{min} \mid \text{max} \quad \text{conditional} \]

Figure 4: Halide expression syntax

\[ R ::= r\text{dom}(r_1 = I_1; \cdots ; r_n = I_n) \quad \text{rdom} \]
\[ U_0 ::= e \quad \text{pure stage} \]
\[ U ::= R\text{in} (e_1, \ldots, e_n) \leftarrow e \text{if } e_p \quad \text{update stage} \]
\[ B ::= U_0 | B; U \quad \text{func body} \]
\[ F ::= \text{fun } f(x_1, \ldots, x_n) = \{B\} \quad \text{func} \]
\[ D ::= F | D; F \quad \text{definitions} \]
\[ P ::= \text{pipeline}(p_1, \ldots, p_m) = D \quad \text{pipeline} \]
\[ Z ::= P(z) \quad \text{realization} \]
\[ z ::= \langle (I_1, \ldots, I_n), (c_1, \ldots, c_m) \rangle \quad \text{input} \]

Figure 5: Halide algorithm syntax

The expression language on its own can only produce errors of kind \( \varepsilon_{\text{arith}} \) (e.g. division by zero). \( \varepsilon_{\text{rdom}} \) captures errors preventing ordinary execution of rdoms (§4.4). Lastly, \( \varepsilon_{\text{mem}} \) are memory errors, which do not occur in the algorithm semantics. They are only possible in the target language (§5.2), but are prohibited by theorem 3.4.

### 4.2 Algorithm Syntax

Our formalization of Halide (syntax in Figure 5) focuses on the fundamental issues at play: pure definitions, separable updates, and imperative updates. Along with pointwise evaluation, these are the primary constructs that govern the structure of computation.

### 4.3 Algorithm validity rules

Halide algorithms must adhere to several non-standard restrictions. This first rule constrains the use of pure variables to facilitate flexible scheduling decisions.

**Definition 4.2** (Syntactic separation restriction). Let \( f \) be a func given by \( \text{fun } f(x_1, \ldots, x_n) = \{U_0; \cdots ; U_m\} \). The syntactic separation restriction states that for all pure vars \( x_i \) and all stages \( U_j \), if \( x_i \) occurs anywhere in \( U_j \) then all accesses in \( U_j \) of the form \( f[e_1, \ldots, e_n] \) must have \( e_i \equiv x_i \).

This rule is so subtle that the authors often misstate it, yet it is critical to the correctness of many scheduling directives and metatheory claims, so we show a few examples.

It might be tempting to write an in-place shift using the following func definition:

\[ \text{fun } f(x) = \{g(x); (x) \leftarrow f(x + 1)\} \]

But such an update diverges on \( f \)'s unbounded domain since \( f(0) \) would need to first compute \( f(1) \), which would need to compute \( f(2) \) and so on. Thus such updates are disallowed by 4.2. It is also disallowed to use
the variable in some places, but not others, as in:

\[
\text{fun } f(x) = \{ g(x); \text{rdom}(r = (0, 3)) \in (x) \leftarrow f(x) + f(r) \}\]

The reason here is that, viewed as an in-place update to the values of \( f \), the update cannot be applied uniformly across the entire dimension \( x \). On the other hand, a definition like

\[
\text{fun } f(x) = \{ 0; \text{rdom}(r = (0, 3)) \in (x) \leftarrow f(x) + g(x) + g(r) \}\]

is perfectly legal since the restriction only applies to the func whose update stage is being defined. At this point in the algorithm, all of \( g \)'s values are known, so there is no hazard. The intuition for this rule is that updates that reference pure variables should augment the previous stage in a mathematically coherent way.

The syntactic separation restriction extends the notion of purity to stage dimensions, which need not reference all of the func’s pure variables.

**Definition 4.3** (Pure/reduction dimensions). For any pure variable \( x_i \) and stage \( U_j \), it is said that \( i \) is a pure dimension in stage \( j \) if \( x_i \) appears in \( U_j \). Dimensions which are not pure are called reduction dimensions.

Certain expressions in Halide are not allowed to refer to variables in order to keep scheduling flexible and sound. Such expressions are called startup expressions to reflect the fact that they are constant from the point of view of any func in the program.

**Definition 4.4** (Startup expression). In a given pipeline \( P \) with parameters \( p_1, \ldots, p_n \), an expression \( e \) is a startup expression iff \( e \) contains no func references and any variable \( v \) occurring in \( e \) is identically one of \( p_i \) for some \( i \).

Finally, we can give the definition of program validity.

**Definition 4.5** (Valid program). A program \( P \in \text{Alg} \) is valid if the bounds of all \text{rdoms} are startup expressions, the names of all funcs are unique, the names of all pure variables within each func are unique, and the names of reduction variables within a single stage are unique. The first stage of every func may not include a self-reference. All stages must obey definition 4.2. Lastly, common type checking rules for expressions (eg. function arity matching) must be respected.

### 4.4 Algorithm Semantics

The purpose of a Halide algorithm is to define the value of every point in every func (Figure 6). Evaluation proceeds pointwise with no need to track bounds. Funcs are evaluated by substitution (Func-Eval) as standard for function calls. Compared to the target language, which precomputes values of Funcs as if they are arrays, this evaluation-by-substitution is lazy.

While this laziness lets us avoid reasoning about bounds, it makes the semantics of the comparatively eager RDoms construct more complicated. How do we update a seemingly pure Func? To resolve this tension we simply unroll RDoms (RDom-Eval) into sequences of point updates when and as they are encountered.

These simple updates can then be thought of as shadowing the previous func definition, similar to the functional definition of stores used by most operational semantics for imperative languages (Update-Eval). If the lookup point and update point coincide, then the update rule is substituted, otherwise the existing value is used.

Lastly, we state that all algorithms terminate.

**Lemma 4.6** (Algorithms terminate). Given any algorithm \( P \in \text{Alg} \) and input \( z \), the output of \( P(z) \) can be determined in a finite amount of time.

This follows the intuition that Halide pipelines are defining mathematical objects by supplying formulas to compute the values.

**Proof.** Since \text{rdom} extents are startup expressions and no infinity value exists in the expression language, there is no way to loop infinitely. Halide does not have any facility for recursion, and so is not Turing-complete. These facts together ensure termination. 

\[\square\]
5 Target Language

In this section, we describe the target language (IR) to which the base Halide algorithm compiles. Programs in this language have a defined execution order (which is modified by the schedule) and is similar to classic imperative languages. It uses the same expression language as the algorithm language and has the same semantics for all expressions, save func accesses, which become references to memory.

5.1 Syntax

Figure 7 presents the abstract syntax for the Halide IR. This language comes in two variants: with holes (\(T_{\text{tgt}}^\star\)) and without holes (\(T_{\text{tgt}}\)). The lowering algorithm given in Section 6 introduces holes that will later be filled by \textit{bounds inference} which is described in Section 7. The main difference between this language and similar imperative languages is that loops are restricted to range-based for loops which can be marked for parallel traversal. The statements \texttt{label} and \texttt{compute} have no actual run time behavior, but are instead used as handles by scheduling and bounds inference.

5.2 Semantics

In Figure 8 we give small-step semantics for the target imperative language. Note that \(\Sigma\) is an \textit{environment} for loop variables and let bindings and \(\_\) is the \textit{store} or heap in which memory is allocated.

The first major difference is that for loops are given as \textit{ranges}, with a \textit{minimum} and a \textit{length}. If the length is negative, or either is an error expression (only \(e_{\text{arith}}\) is possible), then the program steps immediately to a special \textit{error} state.

Second, note that the Alloc rule updates the store \(\_\_\) with a mapping from the \textit{symbolic name} of the func to a pair of (1) a partial function \(\hat{f}\) (initially arbitrary) that records the values and (2) the bounds that were stated at allocation time. The notation \(\overline{I_k}\) is shorthand for \((c_{k_{\min}}, c_{k_{\max}})\).
\[
\begin{align*}
\tau &::= \text{serial} | \text{parallel} \quad \text{execution order} \\
s &::= \text{nop} \quad \text{no operation} \\
&\mid \text{assert } e \quad \text{assertion} \\
&\mid s_1 ; s_2 \quad \text{sequencing} \\
&\mid \text{allocate } (f)(I_1, \ldots, I_n) \quad \text{allocate buffer} \\
&\mid a \leftarrow e \quad \text{update buffer} \\
&\mid \text{if } e_1 \text{ then } s_1 \text{ else } s_2 \quad \text{branching} \\
&\mid \text{let } x = e \text{ in } s \quad \text{let binding} \\
&\mid \text{compute } f \text{ on } (I_1, \ldots, I_n) : s \quad \text{compute label} \\
&\mid \text{label } \ell : s \quad \text{statement label}
\end{align*}
\]

\[
P ::= \text{pipeline}(p_1, \ldots, p_n) : s \quad \text{pipeline}
\]

\[
e ::= \ldots | ? \quad \text{Tgt}^2 \text{ expr}
\]

Figure 7: Halide IR syntax. Expressions are the same as before (Figure 4), but are augmented with holes for Tgt$^2$.

The next two rules define assigning to a point in a func in the store and reading from a func in the store. Assignment is modeled by shadowing the old value, i.e. by redefining the mapping of $f$ in $\sigma$ to a new partial function $\hat{f}'$ which agrees with $\hat{f}$ everywhere except at the point being updated. Reading is simply a matter of evaluating the stored function. The predicate $\text{InBounds}(f[\vec{c}],\sigma)$ checks the fully evaluated point $\vec{c} \equiv (c_1, \ldots, c_n)$ against the bounds stored in $\sigma(f)$.

Note that writing out of bounds stores $\varepsilon_{\text{mem}}$ everywhere in the buffer, thereby invalidating it. Recall that theorem 3.4 states that neither will ever occur in target language programs that were produced by running bounds inference on a lowered and scheduled algorithm.

### 6 Lowering

Halide algorithms are compiled to target language programs with holes by the lowering function $\mathcal{L}$. This function is defined in Figure 9. The lowering function creates a top-level compute statement for every func in the program. In that compute statement are a sequence of nested loops surrounding assignments implementing the formula for each stage in the algorithm. Pure dimensions which do not appear in a stage are not lowered to loops, and reduction domains appear as innermost loops.

The lowering function also introduces labels for each func and stage that can be later used for scheduling. In our treatment, labels are arbitrary, uninterpreted symbols. We use the names of the funcs and number the stages here. The scheduling directives identify loops via the following scheme.

**Lemma 6.1** (Loop naming). Given a valid algorithm $P \in \text{Alg}$ and a valid schedule $T \in \text{Sched}$, any for loop in $S(T, \mathcal{L}(P))$ is uniquely identified by (1) the func, (2) the specialization (or lack thereof), and (3) the stage to which it belongs, as well as (4) the name of its induction variable.

**Proof.** Just after lowering, this is true by construction. Each scheduling directive will preserve this invariant. \qed

Specializations do not exist in initially lowered programs, but can be introduced by scheduling (see §8.1). If a func is not specialized, that data can be regarded as 0. The loop naming lemma lets us talk about the loops for a given stage, or for a given loop, the stage, specialization, and func to which it belongs.

**Lemma 6.2** (Dominance). Given a valid algorithm $P \in \text{Alg}$ and a valid schedule $T \in \text{Sched}$, the program $P' = S(T, \mathcal{L}(P))$ has the following property. If a func $g$ statically depends on another func $f$, then the compute statement for $f$ dominates the compute statement for $g$. Furthermore, the allocate statement for any func $f$ dominates the compute statement for $f$.

\footnote{In the practical system, all funcs are inlined into their consumers by default, but for the sake of formal reasoning, we take the opposite convention.}
Figure 8: IR structural semantics
\[ L_{\text{pipeline}}(p_1, \ldots, p_n) : F_1; \ldots; F_m = \text{pipeline } p_1, \ldots, p_n : L_F (F_1); \ldots; L_F (F_m) \]

\[ L_P(\text{fun } f(x_1, \ldots, x_n) = B) = \begin{cases} \text{allocate } f(\min_{x_1}, \min_{x_1}, \ldots, \min_{x_n}) \\ \text{label } f : \\ \text{compute } f \text{ on } (\min_{x_1}, \min_{x_1}, \ldots, \min_{x_n}) : \\ L_B(f(x_1, \ldots, x_n), B) \end{cases} \]

\[ L_B(f(x_1, \ldots, x_n), U_0; \ldots; U_m) = \begin{cases} \text{label } s_0 : L_U(f(x_1, \ldots, x_n), 0, U_0) \\ \vdots \\ \text{label } s_m : L_U(f(x_1, \ldots, x_n), m, U_m) \end{cases} \]

\[ L_U(f, \bar{x}, i, \text{Rin } (e_1, \ldots, e_n) = \epsilon \text{ if } e_P) = \begin{cases} L_P(f, \bar{x}, (e_1, \ldots, e_n)), \\ L_R(R, \text{if } e_P \text{ then } f[e_1, \ldots, e_n] \leftarrow \epsilon) \end{cases} \]

\[ L_R(\text{rdom}(), s) = s \]

\[ L_R(\text{rdom}(r_1 = I_1, \ldots, r_n = I_n), s) = \begin{cases} L_R(\text{rdom}(r_2 = I_2, \ldots, r_n = I_n), \text{for } r_1 \text{ in } I_1 \text{ do }) \\ \text{if } x_1 \equiv e_1 \end{cases} \]

\[ L_P(f(x_1, \ldots, x_n), (e_1, \ldots, e_n), i, s) = \begin{cases} L_P\left( f(x_2, \ldots, x_n), (e_2, \ldots, e_n), i, \text{for } x_1 \text{ in } (\min_{f(0,0,i-1)} \ldots \max_{f(0,i)}), \text{do } s \right) \\ L_P(f(x_2, \ldots, x_n), (e_2, \ldots, e_n), i, s) \text{ otherwise} \end{cases} \]

Figure 9: Lowering Halide algorithm to IR with default eager schedule.

**Proof.** Just after lowering, this is true by construction. Each scheduling directive will preserve this invariant. □

Lowering introduces a set of **bounds holes** to be filled by the **bounds inference** oracle described in Section 7. Bounds holes are indexable by the following data.

**Definition 6.3** (Bounds hole). A bounds hole is an entity in the expression language of Tgt that stands in for a hole-free expression. A bounds hole contains the following data:

1. Whether it is an allocation hole (**mem**), a compute hole (**cpu**), or a loop hole (**for**).
2. Whether it represents a minimum (**min**), an interval, or the length (**len**).
3. If it is an allocation or compute hole, to which func and dimension it belongs.
4. If it is a loop hole, the name of the loop to which it belongs.

Across specializations, the last stage is defined to share a common bounds hole. Those holes may be specified by only the **cpu** label; min or length; func; and dimension.

## 7 Bounds Inference

In previous work on Halide, bounds inference is discussed in terms of an algorithm used to fill the holes introduced by lowering. This algorithm works by propagating symbolic intervals backwards through the dataflow. Improvements to the compiler regularly change the results of this algorithm, resulting in an unstable definition in practice.

In order to abstract over these changing bounds inference algorithms, we pose bounds inference as a **synthesis** problem via an oracle query. While the resulting satisfiability problem is undecidable, this definition provides previously underformulated soundness conditions for any bounds inference algorithm.

Recall from definition 6.3 that there are three types of bounds in Halide. These are the allocation bounds, the loop bounds, and the compute bounds.
Formally,

\[ B(\text{pipeline}(p) : s) = \forall p_k. B(s) \]
\[ B(\text{nop}) = \text{true} \]
\[ B(\text{label } \ell : s) = B(s) \]
\[ B(s_1; s_2) = B(s_1) \land B(s_2) \]
\[ B(\text{let } v = e \text{ in } s) = (B(s))[e/v] \]
\[ B(\text{if } e \text{ then } s_1 \text{ else } s_2) = (e \implies B(s_1)) \land (\neg e \implies B(s_2)) \]
\[ B(\text{for } v \text{ in } I \text{ do } s) = \forall v \in I. B(s) \]
\[ B(\text{assert } e) = e \]
\[ B(\text{allocate } f(\ldots)) = \forall i. \text{Cpu}(f)_i \subseteq \text{Mem}(f)_i \]

\[ B(\text{compute } f \text{ on } (\ldots) : s) \]
\[ = (\forall i, j, k. \text{Cpu}(f)_k \subseteq \text{Loop}(f, i, j)_k) \land B(s) \]
\[ B(f[e_1, \ldots, e_n] \leftarrow \ldots g[e'_1, \ldots, e'_m] \ldots) = e_i \in \text{Mem}(f)_i \land e'_i \in \text{Mem}(g)_i \]
\[ \land e_i \in \text{Cpu}(f)_i \implies e'_i \in \text{Cpu}(g)_i \]

Figure 10: Bounds oracle query extraction

The compute bounds define regions over which the points in the buffer must have values that agree with those defined by the original algorithm. The allocation bounds enclose the compute bounds, and the actual points that are touched sit in between the two. The loop bounds must ensure that all of the points in the compute bounds are actually computed. To ensure correctness, if a point being computed lies in the compute bounds, then all of the accesses on the right hand side of the assignment must be in the compute bounds of their funcs. In short, what happens outside the compute bounds stays outside the compute bounds. This gap can then be exploited by overcompute strategies during scheduling (§8.2).

The algorithm for extracting the bounds constraint for a program \( P \in \text{Tgt} \) is shown in Figure 10. The extraction traverses the AST of the program and translates every statement into a logical condition with holes for which any satisfying completion implies the following.

**Definition 7.1 (Bounds sets).** We define a few functions for the purpose of defining the bounds extraction in Figure 10.

\[ \text{Mem}(f)_i = \begin{bmatrix} \text{mem.min}_f & \text{mem.len}_f \\ \text{mem.min}_f & \text{mem.len}_f \end{bmatrix} \]
\[ \text{Cpu}(f)_k = \begin{bmatrix} \text{cpu.min}_f & \text{cpu.len}_f \\ \text{cpu.min}_f & \text{cpu.len}_f \end{bmatrix} \]

Lastly, \( \text{Loop}(f, i, j)_k \) refers to the set of values dimension \( k \) takes on in the write side of the assignment in stage \( j \) of specialization \( i \) of func \( f \). This is defined in terms of the expression appearing in that position.

**Definition 7.2 (Bounds oracle query).** Let \( P \in \text{Alg} \) be an algorithm and let \( T \in \text{Sched} \) be a schedule for it so \( S = S(T, \mathcal{L}(P)) \). Then a query to the bounds oracle \( \mathcal{O} \) consists of (1) the predicate \( p = B(S) \) and (2) the lexical scopes \( K(S) \) allowed for every hole in \( p \).

The oracle responds with some list of hole substitutions \( V \in \mathcal{O}(p, K) \) that is compatible with \( S \). Hence, the set \( \mathcal{BI}(P) = \{ S[V] \mid V \in \mathcal{O}(B(S), K(S)) \} \).

Here we investigate the structure of solutions to bounds inference.

**Definition 7.3 (Narrowing program).** Let \( P_1, P_2 \in \mathcal{BI}(S) \) where \( S \) is a program scheduled from some algorithm. We say that \( P_1 \leq P_2 \) iff for all inputs \( z \), the every execution of any compute, allocate, or for statement in \( P_1(z) \) has a corresponding execution in \( P_2(z) \) and the bounds \( I_1 \) for \( P_1(z) \) are contained in the bounds \( I_2 \) for \( P_2(z) \). (Here “corresponding” means that the two statements are at the same lexical site, executing with identical environments \( \Sigma \).)
Lemma 7.4 (Narrowing executions match). Let $P_1 \leq P_2$ as above. Then after the execution of corresponding compute statements for some func $f$ on intervals $I_1$ and $I_2$, the contents of the buffer for $f$ agree on $I_1$.

Proof. Let $SI(P, z)$ denote the set of assignment executions in $P(z)$. By definition, $SI(P_1, z) \subseteq SI(P_2, z)$. Let $SI(P_2, z)|_{I_1}$ be the set of assignment executions within the compute statement of $P_2(z)$ which write to a point of $f$ in $I_1$. Then $SI(P_1, z) \supseteq SI(P_2, z)|_{I_1}$ (by the CPU $\subseteq$ Loop constraint of $B$). Thus there are no writes to values in $I_1$ in $P_2$ that do not also occur in $P_1$. Furthermore, all of these computed values depend only on their values computed in both programs. This follows from the assignment rule of $B$.

Lemma 7.5 (Memory safety). All bounds query results $P' \in BI(S)$ for $S$ any scheduled program, are memory safe.

Proof. First recall that by lemma 6.2, every access to a func $f$ is dominated by the allocate statement for $f$. The rule for accesses in $B$ explicitly requires that every referenced point is contained in the allocation bounds (Mem) of the associated func. Thus $P'$ will always satisfy the InBounds condition.

Note that if an expression appearing in an access is either potentially erroring or unbounded then there is no possible bounds query result. In this case, the lemma 7.5 holds vacuously. We will now begin to prove that lowering is confluent with the algorithm.

Lemma 7.6 (Compute bounds confluent). Let $P \in Alg$ and $P' \in BI(L(P))$. Let $f$ be a func in $P$. Assume that all of the points in compute bounds of funcs $g$ preceding $f$ are confluent with $P$. Then the compute statement for $f$ computes values confluent with $P$.

Proof. Recall from the definition of $B$ that each of $f$’s stages’ loop bounds cover the compute bounds. Lemma 6.2 ensures that the func is realized before it is read.

Thus, if $f$ consists of only a pure stage, we are done because the static lack of self-reference and reduction dimensions makes each assignment in that stage completely independent of every other. The assignment exactly matches the algorithm’s expressions and only reads from compute bounds by assumption.

On the other hand, if $f$ has $n$ stages and $P'$ satisfies the claim, we argue that adding another stage $s$ to $f$ preserves the claim. The definition of $B$ ensures that every access in $s$ is included in the allocation bounds and assignments writing to point in the compute bounds read only within the compute bounds of other funcs and $f$. By the induction hypothesis on stages, the values in the compute bounds of the buffer for $f$ (and all other funcs) are correct just before $s$ runs.

Recall the structure of the loop nest for $s$. The outermost for loops correspond to pure dimensions and range over bounds holes, which are constrained to cover the compute bounds of $f$. The innermost for loops implement any reduction domain in the stage and have bounds supplied by the algorithm. If the stage is guarded by a condition, it is included within the innermost for loop. Finally, a single assignment statement corresponding to the update rule for the stage is the innermost statement.

We need to show that the code produced by lowering for $s$ will compute confluent values for $f$. Consider a point $p$ in the compute bounds of $f$ after the stage runs. The stage $s$ separates pure dimensions from reduction dimensions of $p$. Let $x_p$ be the assignment to pure dimensions induced by $p$. Consider the definition of $s$ in the algorithm. It consists of a series of simple updates after unrolling the rdom all of which share values $x_p$ in common. Now, observe the iteration of the pure loops of $s$ in the target language which coincide with $x_p$. This iteration is guaranteed to occur by the covering of the compute bounds by the loop bounds. The content of this iteration is a sequence of assignments corresponding to the unrolled rdom. Lastly, any other $x'_p \neq x_p$ touches no memory in common with this iteration because of syntactic separation (def. 4.2).

Lemma 7.7 (Lowering is sound). $S = L(P)$ is confluent with $P$ for all $P \in Alg$.

Proof. Let $P' \in BI(S)$ and let $z$ be any input. If $P(z)$ contains an error, then we are done. Since lowering does not create any assertion statements, failing one is not possible. All lowered programs trivially respect dominance. Finally the argument in lemma 7.6 applies inductively over the lowered code.

The proof of correctness for the remainder of of theorem 3.5 will follow by induction from the proofs for each scheduling directive in section 8.
8 Scheduling Language

We formalize scheduling by directly mutating programs in $\text{Tgt}^\ell$. Because some directives — like split — are incompatible with other directives, we assume that all schedules are ordered into phases as indicated in Figure 11. Scheduling directives use loop names as given by lemma 6.1 to determine their targets.

In Figure 12 we show the IR transformations for each scheduling directive. In each subsequent section, we describe each phase, enumerate its restrictions, and prove its safety.

8.1 Specialization Phase

Certain scheduling decisions may be more or less efficient, depending on program parameters. For instance, simpler schedules tend to work better for small output sizes.

This phase addresses this issue by duplicating an existing func’s code for each of $n$ conditions. These specializations introduce labels that allow later scheduling directives to operate differently on each one. In our formal system, schedules may give at most one specialization directive per func.

Lemma 8.1 (Unique active specialization). Given algorithm $P \in \text{Alg}$ and a schedule $T \in \text{Sched}$, let $P' \in \text{BI}(S(T, \mathcal{L}(P)))$. Then for any input $z$, $P'(z)$ will evaluate exactly one specialization for any given func $f$.

Proof. If $f$ has no specializations, then its only compute statement is considered the default specialization. Valid schedules have at most one specialization directive per func, so there is one block of if-then statements for each func, each containing a compute statement for $f$. Since the conditions of those branches are startup expressions, the input $z$ determines which one will be taken every time the block is encountered.

Theorem 8.2 (Specialization is sound). Let $P \in \text{Alg}$ be a valid algorithm and let $S_i \in \text{Tgt}^\ell$ be the result of lowering $P$ and applying scheduling directives up through this phase. Let $s$ be a scheduling directive in this phase, then $S_{i+1} = S(s, S_i)$ is confluent with $P$.

Proof. Let $z$ be a valid input for $P$, $P' \in \text{BI}(S_i)$, and $P'_{i+1} \in \text{BI}(S_{i+1})$. If $P(z)$ contains an error, we are done; and no assertions are present for until the bounds phase; so we may assume $P \cong z P'_i$. By assumption, $s$ is a specialization of some func $f$. Now by lemma 8.1, exactly one branch of $f$ is taken in any execution $P'_{i+1}(z)$. This preserves producer domination as required by lemma 6.2.

---

Figure 11: The Halide scheduling language. The $s$ definition is grouped by phase of scheduling (presented in order).
Figure 12: Scheduling directives over the IR
8.2 Loops Phase

Halide provides several standard loop transformations to change the order of computations. A loop can be split into two nested loops, two nested loops can be fused into a single loop, two nested loops may be swapped, and loops may be traversed in parallel.

Some loop directives apply only to pure loops, a manifestation of pure dimensions in the target IR.

**Definition 8.3 (Pure loop).** Let \( v \) be the loop variable for some for loop in a program \( P \in \text{Tgt}^2 \). We say \( v \) and its associated loop are pure if \( v \) is identically one of the pure dimensions of its associated func, if it is the result of splitting a pure loop, or if it is the result of fusing two pure loops together. We letter the iteration variable of pure loops \( x \). All other loops are reduction loops, lettered \( r \).

We may split a loop \( l \) by \( e \) into an outer loop iterated by \( x_o \) and inner loop by \( x_i \), which runs over the split factor \( e \). This division may produce a remainder, which is handled in the practical system by choice of a tail strategy: (1) guarding the body with an if; (2) overcomputation of the func (affecting bounds); (3) shifting the last loop iteration inwards, causing recomputation. The latter two are only allowed on pure stages.

Two nested loops can be fused together into a single loop whose extent is the product of the original extents, provided both loops are pure or both are reduction loops. This is approximately an inverse to the split directive, and is useful for controlling the granularity of parallelism. Nested loops can be swapped as long as the swap does not reorder two loops over reduction variables \(^6\). Finally, each pure loop can also be traversed in either serial or parallel order \(^7\).

All variable names introduced by directives must be new, unique and non-conflicting. Now we can argue that programs are safely transformed by these directives.

**Theorem 8.4 (Loop phase is sound).** Let \( P \in \text{Alg} \) be a valid algorithm and let \( S_i \in \text{Tgt}^2 \) be the result of lowering \( P \) and applying scheduling directives up through this phase. Let \( s \) be a scheduling directive in this phase, then \( S_{i+1} = \mathcal{S}(s, S_i) \) is confluent with \( P \).

**Proof.** Theorem 8.2 ensures that \( S_i \) is confluent as long as we have not yet reached this phase. So we may inductively assume confluence before issuing any such \( s \). As before, \( s \) does not introduce assertions, so we need only assess output equivalence. Each directive \( s \) operates locally on loops in a single stage, so therefore we need only show that the transformation of this stage is observationally equivalent; i.e. the state of the buffers before and after the stage is the same within the compute bounds.

Suppose \( s = \text{split}(\ldots, v, v_o, v_i, e^{\text{fac}}) \). We may check that the values \( v \) ranges over are unchanged; this means that \( B(S_i) \) is logically equivalent to \( B(S_{i+1}) \). Since the scopes of holes have also not changed (\( K(S_i) = K(S_{i+1}) \)), the sets of bounds query results are identical. Thus there is a bijection between programs \( P'_i \in \text{BI}(S_i) \) and \( P'_{i+1} \in \text{BI}(S_{i+1}) \), s.t. corresponding holes have been filled with identical expressions. These two programs execute the same statement instances in the same order; therefore the effect on buffers is identical.

The argument for the fuse directive proceeds analogously, as do the arguments for the overcomputation and inward-shifting split strategies that apply only to single-stagefuncs, which are adjusted for the expanding compute bounds and identically recomputed points, respectively.

For swap and traverse, consider the loop nest for the stage of \( f \). Analogously to the proof of lemma 7.6, syntactic separation ensures that distinct iterations of pure loops access \( f \) at disjoint sets of points. Since split and fuse preserve the sets and order of accesses as they introduce new loops, two writes agree on their pure dimensions iff the pure loop variable values agree.

But this means that two assignments touch the same memory of \( f \) only if the values of the pure loops are the same. Since traverse parallelizes over a single pure loop, no two tasks touch the same memory. And swap interchanges two pure loops, so the writes which do touch memory in common are not interchanged with respect to one another.

---

\(^6\)The practical Halide system allows swaps between reduction variables predicated on a successful commutativity check \([35]\).

\(^7\)The practical Halide system also adds unrolled, and vectorized traversal orders, but these are not formalized here.
8.3 Compute Phase

To narrow the scope of computation, the compute statement for a func $f$ may be moved from the top level to just inside any loop as long as the compute statement continues to dominate all external accesses to $f$.

The closer a producer is computed to its consumer, the less of the producer needs to be computed per realization. The expectation is that bounds inference will use the additional flexibility granted by the additional loop iteration information to derive tighter bounds. This directive therefore controls how much of a Func to compute when it is realized.

**Lemma 8.5 (Compute phase is sound).** Let $P \in \text{Alg}$ be a valid algorithm and let $S_i \in \text{Tgt}^2$ be the result of lowering $P$ and applying scheduling directives up through this phase. Let $s$ be a compute-at directive moving the func $f$ to some location $\ell_g$. Then $S_{i+1} = S(s, S_i)$ is confluent with $P$.

**Proof.** Let $P'_i \in \text{BI}(S_i)$ and let $\{P''_i\}$ be the set of programs resulting from applying $s$ to $P'_i$. Let $P'_{i+1} \in \text{BI}(S_{i+1})$. First observe that any $P''_i$ computes the same values as $P'_i$ because the compute statement for $f$ computes all of the points ever needed in $P''_i$ and dominates all of its consumer funcs. This also implies that $P''_i \in \text{BI}(S_{i+1})$.

Now suppose $P'_{i+1}$ is not one of the $P''_i$. Then for each $z$, if $P(z)$ does not contain an error, then there is some $P''_i \geq P'_{i+1}$ (see definition 7.3). By lemma 7.4, $P''_i$ computes the exact same values of $f$ on the narrower range, which bounds inference guarantees is sufficient for confluence.

8.4 Storage Phase

Each Func is tied to a particular piece of memory when it is realized. Halide offers some control over how much memory a Func occupies during the run of a pipeline. The store-at directive (analogous to compute-at above) moves the allocation statement to just inside any loop such that the allocation still dominates all accesses of the Func it allocates.

Bounds inference is then free to choose a smaller, more precise size for the allocation based on the code that follows, and the particular values of the variables of the loops that enclose it.

**Lemma 8.6.** Let $P \in \text{Alg}$ be a valid algorithm and let $S_i \in \text{Tgt}^2$ be the result of lowering $P$ and applying scheduling directives up through this phase. Let $s$ be a store-at directive, then $S_{i+1} = S(s, S_i)$ is confluent with $P$.

**Proof.** Lemma 7.5 required only dominance of the allocate statement over all accesses to the associated Func for correctness. This is preserved by definition.

8.5 Bounds Phase

Additional domain knowledge might allow a user to derive superior bounds functions than those inferred. Halide provides directives to give hints to the bounds inference oracle just before querying it.

The first two directives, bound and bound-extent, assert equality of bounds holes to provided startup expressions. The third directive, align-bounds, adds assertions that constrain the divisibility and position of the window. The minimum is constrained to have a particular remainder modulo a factor which is declared to divide the extent. These assertions affect the bounds inference query such that the inferred computation window will expand to meet these requirements.

**Theorem 8.7 (Bounds phase is sound).** Let $P \in \text{Alg}$ be a valid algorithm and let $S_i \in \text{Tgt}^2$ be the result of lowering $P$ and applying scheduling directives up through this phase. Let $s$ be a scheduling directive in this phase, then $S_{i+1} = S(s, S_i)$ is confluent with $P$.

**Proof.** The directives in this phase only mutate programs by adding assertions to them, the only side effect of which is to transition to an error state. Thus in any non-erroring execution of any $P_i \in \text{BI}(S_i)$, there is some $P_i \in \text{BI}(S_i)$ whose behavior exactly matches $P_{i+1}$.
8.6 Practical directives

Halide provides many more scheduling directives that are out of scope for this paper. It has directives for assigning loops to coprocessors like GPUs and the Hexagon DSP; and directives for prefetching and memoization. Some of the most esoteric directives (compute-with and async) may require more substantial modification or extension of these semantics.

9 Related Work

The computational and scheduling models of Halide have evolved through a series of extensions and generalizations [32, 30, 31, 35]. Halide builds on the idea of explicit control over compiler transformations developed earlier in many script- or pragma-based compiler tools in HPC [15, 17, 42, 20, 9], and the definition of parametric spaces of optimizations in SPIRAL [18]. A growing family of high performance DSLs since the introduction of Halide have directly adopted the concept of a programmer-visible scheduling language [3, 10, 38, 29, 43, 39, 25]. The Polyhedral loop optimization community has absorbed and extended these ideas in its own context [40, 41, 1, 2]. None of these languages and systems, however, have been described and analyzed formally. The recent TeML language is defined in a formal style, but its transformation language is not correctness-preserving, and it does not make safety guarantees or suggest how such a metatheory might be formulated [36]. Stronger equivalence properties have been proven in recent work on type-based decompositions into algorithmic skeletons [5].

Halide’s algorithm language is closely related to both array languages [26, 8, 7, 23, 6], and image processing DSLs [24, 34]. Its computational model is most closely related to that of the lazy functional image language Pan [16]. Bounds inference is related to array shape analyses and type systems [27, 28, 22]. Our treatment of bounds inference here is formulated as a constraint-based program synthesis problem [19].

The correctness of many compiler transformations has been treated in the context of verified compilers like CompCert [4, 37]. This goal, however, is inherently more ambitious than ours: we analyze the safety and correctness of operators in a language designed (albeit informally) with roughly these goals in mind, while optimizations in CompCert have to preserve every visible program property in the notoriously complex C language.

10 Conclusion

This paper presents the first major step towards formalizing a real-world language with user-controlled scheduling, covering most essential features of the practical Halide language. Some features are omitted for simplicity, but would be interesting to consider in future work. Atomics, asynchronous producer-consumer parallelism, and mapping to heterogeneous hardware all require a model of concurrency. We also omit some important optimizations from the real system which apply after scheduling and bounds inference (sliding window, storage folding). Finally, we do not model floating point. Defining correctness in floating point computations is inherently difficult, and the real Halide system already makes “fast-math” assumptions (effectively treating floats as reals) by default, which is common in its target domains.

We believe this work already provides a foundation to study this new class of languages with user-controlled scheduling. One major question is how they could incorporate abstraction and module systems. Another is whether alternative bounds inference algorithms, based on our program synthesis formulation, could be useful in practice.

Practical impact

These formalization efforts have also already influenced Halide’s design, and found and fixed bugs where actual and expected behavior differed in significant ways.

**Negative rdom extents.** During the course of research, the authors discovered the behavior for negative rdom extents is not defined in the practical system [14]. By happenstance, negative-extent rdoms are treated
as no-ops in simple cases designed to probe the behavior. However, there could be instances in the compiler where a negative extent is treated as unsigned, which would silently lead to a very long loop.

At the time of writing, the implications of defining any particular behavior for negative-extent rdoms has not been explored, so they are formalized here as errors.

**Realizing outputs with rdoms.** For efficiency, the practical Halide system allows a user to supply their own output buffer, rather than allowing Halide to allocate the memory. These buffers are checked against the inferred allocation bounds. However, this same system was used to implement the common interface that requests only a compute window. This led to vexing errors on safe output windows [11]. But since adding an identity func after the original output would allow bounds inference to expand the intermediate buffer and copy the requested window to the output, from the perspective of the user, identity functions could be *impure*. This issue was fixed by extending the interface to match the expected behavior, formalized here.

**Arithmetic error semantics.** In this paper, numerical operations report errors by returning special values, rather than crashing the program. This is also the case for IEEE 754 floating point. However, it is *not* the case for integer division on x86. Such errors include division by zero and multiplication of the most negative (two’s complement) value by -1.

When split rounds up, it is assumed that the values between the compute and allocation bounds will not affect the final output. However, one of the uninitialized values used in an integer computation in this gap can cause a crash [13]. At the time of writing, fixes are being discussed.

**Compute-with directive.** *Compute-with* is a scheduling directive intended to zip together the loops of independentfuncs for the purpose of increasing parallelism. However, the implementation was only tested on funcs with a single pure stage, despite an interface and an attempt to support fusion of update stages. Consequently, scheduling is currently broken for programs using this feature.

We have identified a potential fix that involves explicitly adding fused groups to the algorithm language and topologically sorting them before lowering [12]. However, more work is needed to formalize the feature and patch the compiler, while making any consequent adjustments to the semantics.

**Acknowledgements**

We thank Andrew Adams and Daan Leijen for their helpful conversations about Halide’s implementation and language semantics, respectively.

**References**


[14] Halide Contributors. What does it mean to have an RDom with a negative extent? • Issue #4385 • halide/Halide, Nov 2019.


