

HELIX: From Math to Verified Code

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Vadim Zaliva

A.S. Electrical Engineering, Kiev College of Radio Electronics

B.S. Computer Aided Design Systems, Ukraine State Academy of Light Industry

M.S. Computer Science, Colorado State University

Carnegie Mellon University

Pittsburgh, PA

December 2020

©Vadim Zaliva, 2020

All Rights Reserved

For my father.

Acknowledgments

There are so many people who have supported me in my work on this thesis over the years that I am in peril of forgetting to mention some of them. I am grateful for everyone’s support, even if I have neglected to mention their names below.

First of all, I would like to thank my committee members - Dr. Steve Zdancewic, Dr. Corina Pasareanu, Dr. Limin Jia, and my advisor Dr. Franz Franchetti for being a part of this journey.

The key person without whom none of it would be possible is my advisor Franz Franchetti. He not only chose me to be his student despite my non-typical background and the fact that my research interests were not immediately related to his, but he also gave me an opportunity to find my way by pursuing research directions of my own choosing. He provided steadfast support, advice, and encouragement during all these years. I could not wish for a better advisor than him. An exemplary piece of mentoring advice from him was, “I think you are wrong, but you should try to do this anyway to see it for yourself.” (For the record, I was wrong).

Tze Meng Low, Doru Thom Popovici, Richard Veras, Berkin Akin, Jiyuan Zhang, and Daniele Spampinato from the CMU SPIRAL research group at Pittsburgh always engaged me in stimulating technical discussions, explained to me intrinsic details of SPIRAL system, provided feedback on my presentations and papers, and gave invaluable general academic advice. Our outings at The Porch were highlights of my visits to the main campus.

Second, I would like to thank people at the University of Pennsylvania: Steve

Zdancewic, Yannick Zackowski, Calvin Beck, and Irene Yoon with whom I collaborated very closely during the last few years. They accepted me and made me feel like a member of their research group. This collaboration was a huge contrast and a welcome change to the mostly solitary research I had been doing before, and I greatly enjoyed and benefited from it.

There are many people from the PL community who influenced me and acted as role models. At various conferences and online, I often pestered them with my sometimes naïve questions soliciting their advice. There are too many to list, but to name a few: Andrew W. Appel, Adam Chlipala, Clément Pit-Claudel, Benjamin J. Delaware, and Gregory Malecha. Dr. Lester Ludwig was instrumental in steering me towards the CMU PhD program. Jeremy Johnson from Drexel University provided a much needed kick start at the beginning of my journey. Matthieu Sozeau dedicated an extended amount of his time by inviting me to visit him at the Institut de Recherche en Informatique Fondamentale, which helped me to advance my research and gave me a chance to better understand Coq internals.

My business partner, Alex Sova, gave me the opportunity to pursue my PhD by letting me take an extended sabbatical from our business. Last, but not least, I would like to thank my wife, Tanya, who was patient and supportive during my frequent travels, irregular working hours, and missed vacations and weekends throughout all of these years.

The work in this dissertation could not have happened without support from the Defense Advanced Research Projects Agency (DARPA), National Science Foundation (NSF), US Department of Energy (DOE), and CMU Software Engineering Institute (SEI).

VADIM ZALIVA

Abstract

This thesis presents HELIX, a code generation and formal verification system with a focus on the intersection of high-performance and high-assurance numerical computing. This allowed us to build a system that could be fine-tuned to generate efficient code for a broad set of computer architectures while providing formal guarantees of such generated code’s correctness.

The method we used for high-performance code synthesis is the algebraic transformation of vector and matrix computations into a dataflow optimized for parallel or vectorized processing on target hardware. The abstraction used to formalize and verify this technique is an operator language used with semantics-preserving term-rewriting. We use sparse vector abstraction to represent partial computations, enabling us to use algebraic reasoning to prove parallel decomposition properties.

HELIX provides a formal verification foundation for rewriting-based algebraic code synthesis optimizations, driven by an external oracle. Presently HELIX uses SPIRAL as an oracle deriving the rule application order. The SPIRAL system was developed over the years and successfully applied to generate code for various numeric algorithms. Building on its sound algebraic foundation, we generalize and extend it in the direction of non-linear operators, towards a new theory of partial computations, applying formal language theory and formal verification techniques.

HELIX is developed and proven in Coq proof assistant and demonstrated on a real-life example of verified high-performance code generation of the dynamic window safety monitor for a cyber-physical robot system.

Contents

Acknowledgments	iv
Abstract	vi
List of Figures	xii
List of Tables	xiv
List of Listings	xvi
Chapter 1 Introduction	1
1.1 Problem Statement	1
1.2 SPIRAL Overview	3
1.3 HELIX Introduction	4
1.4 Motivating Example	8
Chapter 2 Background	14
2.1 Formal Verification	14
2.2 Mathematical Foundations	15
2.3 Coq Proof Assistant	16
2.3.1 Quick Introduction to Coq	19
2.4 Formal Semantics of Programming Languages	21

2.5	Certified Compilation and Translation Validation	23
Chapter 3	HELIX Formalization	25
3.1	<i>HCOL</i> Language	28
3.1.1	Carrier Type	30
3.1.2	Equality	31
3.1.3	<i>HCOL</i> operators	32
3.1.4	Example	37
3.2	Σ - <i>HCOL</i> Language	37
3.2.1	Modelling Missing Values and Collisions	39
3.2.2	Index Mapping Functions	43
3.2.3	Families of Index Mapping Functions	44
3.2.4	Operator Type	45
3.2.5	Structural Correctness	46
3.2.6	Operator Families	48
3.2.7	Equality	48
3.2.8	Σ - <i>HCOL</i> Operators	49
3.2.9	Sparse Embedding	57
3.2.10	Map-Reduce	58
3.2.11	Relation between <i>HCOL</i> and Σ - <i>HCOL</i>	59
3.2.12	Example	60
3.3	<i>MHCOL</i> Language	60
3.3.1	Memory Model	64
3.3.2	Example	67
3.4	<i>DHCOL</i> Language	68
3.4.1	<i>DHCOL</i> Type Parametrization	69

3.4.2	<i>DHCOL</i> Expressions and Operators	75
3.4.3	<i>DHCOL</i> Evaluation	78
3.4.4	Example	79
3.5	<i>FHCOL</i> Language	81
3.5.1	Example	84
Chapter 4	HELIX Verification	86
4.1	<i>HCOL</i> Breakdown	89
4.1.1	Breakdown Rules as Lemmas	91
4.1.2	Semantics-preserving rewriting	93
4.1.3	<i>HCOL</i> Semantics Preservation Verification Framework	95
4.2	<i>HCOL</i> to Σ - <i>HCOL</i> Translation	97
4.3	Σ - <i>HCOL</i> rewriting	99
4.3.1	Restricted Monoid	99
4.3.2	Example	103
4.4	Σ - <i>HCOL</i> to <i>MHCOL</i> Translation	104
4.4.1	Implementation	104
4.4.2	Proof of Semantics Preservation	104
4.5	<i>MHCOL</i> to <i>DHCOL</i> Translation	106
4.5.1	Implementation	106
4.5.2	Proof of Semantics Preservation	110
4.6	<i>DHCOL</i> to <i>FHCOL</i> Translation	116
4.6.1	Implementation	116
4.6.2	Proof of Semantics Preservation	117
4.7	<i>FHCOL</i> to LLVM IR Translation	117
4.7.1	<i>FHCOL</i> to LLVM IR compiler	119

4.7.1.1	Type Mapping	119
4.7.1.2	Translation Units	120
4.7.1.3	Compiler Organization	121
4.7.2	Compiler Correctness	125
4.7.3	Compiler Testing	130
4.7.4	DHCOL Denotation Semantics	132
4.7.5	Example	135
Chapter 5	Results and Discussion	136
5.1	Implementation	136
5.2	Coverage	139
5.3	Future Research Directions	140
5.3.1	Integer overflow proofs	141
5.3.2	Floating-point proofs	141
5.3.3	Finite-sets proofs	143
5.3.4	LLVM vector instructions	144
5.3.5	Algebraic Theory of Partial Computations	144
5.4	Related Work	144
5.5	Contributions and Lessons Learned	146
5.6	Concluding Remarks	147
Appendix A	<i>DHCOL</i> Big-Step Operational Semantics	149
A.1	The Set of States	150
A.2	Expressions	151
A.3	Operators	157

Appendix B Examples of Generated Code	161
B.1 Dynamic Window Monitor in LLVM IR	161
B.2 Dynamic Window Monitor in C	173
Bibliography	175

List of Figures

1.1	SPIRAL transformation stages	4
1.2	HELIX transformation stages	6
1.3	HELIX application to high-assurance vehicle control	10
1.4	HACMS vehicles	13
2.1	Coq architecture	18
3.1	HPointwise operator	32
3.2	HBinOp operator	34
3.3	Dense vector as sum of four sparse vectors	38
3.4	Dense vector as sum of two sparse vectors	38
3.5	Embed dataflow	49
3.6	Pick dataflow	50
3.7	Scatter dataflow	51
3.8	Gather dataflow	52
3.9	Apply2Union dataflow	54
3.10	Map-Reduce of a Sparse Embedding	59
3.11	Sparse vectors as dictionaries	61
3.12	MApply2Union in <i>MHCOL</i>	62
3.13	Memory model modules	65

3.14	<i>DHCOL</i> modules	74
4.1	HELIX chain of verification	86
4.2	Dynamic Window Monitor dataflow graph in <i>HCOL</i>	91
4.3	HELIX translation validation of SPIRAL <i>HCOL</i> rewriting	96
4.4	Variable resolver example	110
4.5	<i>DHCOL</i> and <i>MHCOL</i> equality relation	114
4.6	<i>FHCOL</i> semantics	119
4.7	A screenshot of the compiler test suite execution	132

List of Tables

3.1	Summary of HELIX languages	26
5.1	Implementation code size	136

Listings

3.1	<code>HPointwise</code> operator definition	32
3.2	Dynamic Window Monitor in <i>HCOL</i>	37
3.3	Exclusive Union Monoid	40
3.4	“Safe” Union Monoid	42
3.5	<code>Vec2Union</code> vector combining operation	43
3.6	<code>SHOperator</code> type	45
3.7	Dynamic Window Monitor in Σ - <i>HCOL</i>	60
3.8	<code>MSHOperator</code> definition	62
3.9	Partial <code>CType</code> module type definition	65
3.10	Memory interface	66
3.11	Memory block interface	67
3.12	Dynamic Window Monitor in <i>MHCOL</i>	67
3.13	<code>CType</code> module type	70
3.14	<code>NType</code> module type	71
3.15	<code>AExpr</code> type	75
3.16	<code>DSHOperator</code> type	76
3.17	Dynamic Window Monitor in <i>DHCOL</i>	79
3.18	Dynamic Window Monitor in <i>FHCOL</i>	84
4.1	Dynamic Window Monitor in <i>HCOL</i> after rewriting	90
4.2	<code>HScalarProd</code> breakdown rule	92

4.3	<code>HEvalPolynomial</code> breakdown rule	93
4.4	<code>HOperator</code> class	94
4.5	<code>HOperator</code> instance for <code>HPointwise</code>	95
4.6	Restricted Monoid	100
4.7	Dynamic Window Monitor in Σ - <i>HCOL</i> after rewriting	103
4.8	Σ - <i>HCOL</i> and <i>MHCOL</i> main semantic equivalence property	105
4.9	<code>MSHCompose</code> operator translation to <i>DHCOL</i>	107
4.10	Variable resolvers	108
4.11	<i>DHCOL</i> module specializations	116
4.12	<i>FHCOL</i> Program definition	120
4.13	Compiler state frame	122
4.14	IR module organization in pseudo-C	126
4.15	IR compiler correctness theorem	128
4.16	<i>DHCOL</i> events	132
4.17	<i>DHCOL</i> semantic equivalence	134
5.1	Finite set proof obligation example	143
B.1	Dynamic Window Monitor in LLVM IR	161
B.2	Dynamic Window Monitor in C (not optimized)	173
B.3	Dynamic Window Monitor in C (optimized)	174

Chapter 1

Introduction

1.1 Problem Statement

The increased dependence of modern society on computer programs combined with the growing sophistication of software systems and hardware architectures poses a significant challenge in keeping computer systems reliable and correct. For example, Boeing's 787 airplane contains about 6.5 million lines of code to operate its avionics and onboard support systems [1]. According to the same source, the premium class automobile contains more than 100 million lines of code. A lot of this code controls mission-critical systems, which, if they malfunction, could endanger human lives. There are multiple reported cases or costly recalls when such bugs were discovered and deemed critical. There are significant ongoing engineering efforts in developing reliable embedded software at such a scale [2].

Most vehicular embedded software is run on one of the Electronic Control Units (ECUs), controlling various car functionality aspects. Lower-end models utilize between 30-50 ECUs, while in high-end luxury cars, this number could be even higher. These ECUs could utilize different Instruction Set Architectures (ISAs),

memory architectures, clock speeds, and data bus widths. This makes the problem of developing software even more formidable. Not only does one have to write and maintain millions of lines of code, but this code will be targeting multiple hardware architectures. Due to price and energy consumption constraints, these embedded systems are typically underpowered compared to modern desktop computers. At the same time, they often control functions that require high-performance computations. For example, an ECU, which controls airbag deployment, must react within 15 to 40 milliseconds [1].

Software for cyber-physical systems could be roughly split into algorithmic and numeric parts. The key requirements for such software are high performance and correctness. The algorithmic part is well studied and known approaches exist for its high-performance implementation and verification. The challenges posed by the numeric part are often underappreciated. For example, converting a mathematical formula describing a robot's dynamics into high-performance machine implementation is not trivial. Proving the correctness of the implementation is even less so. While for some operations, well-known fast and numerically stable algorithms exist; however, when combining such operations into more complex expressions, these properties are not guaranteed to be preserved.

The problem we are trying to address lies on the intersection of high-performance and high-assurance numerical computing. Our goal is to design a system for generating high-performance code for a class of numeric algorithms, useful for practical real-life applications. The system has to be formally verified from top to the bottom providing formal guarantees or correctness of generated high-performance implementation.

1.2 SPIRAL Overview

With the current level of sophistication of hardware architectures, the problem of high-performance implementation of numerical algorithms becomes challenging for manual implementation even when using optimizing compilers and is often solved by specialized code generation systems, such as SPIRAL [3].

Developed over the last 20 years, the SPIRAL system has been used to generate, synthesize, and autotune programs and libraries. It works by translating rule-encoded high-level specifications of mathematical algorithms into highly optimized/library-grade implementations. SPIRAL has been used to formalize a variety of computational kernels from the signal and image processing domain, including graph algorithms, robotic vehicle control, software-defined radio (SDR), and numerical solution of partial differential equations. SPIRAL is capable of generating code for multiple platforms ranging from mobile devices and multicore (desktop and server) processors to high-performance and supercomputing systems [4].

SPIRAL works by transforming the original expression into a series of intermediate languages (as shown at Figure 1.1). The translation steps correspond to different levels of abstraction:

1. Mathematical formula
2. The dataflow (OL)
3. The dataflow with implicit loops (Σ - OL)
4. Imperative program (i -Code)
5. Mainstream programming language code: (C Program)

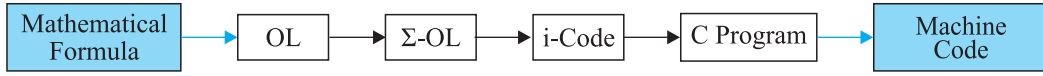


Figure 1.1: SPIRAL transformation stages

The dataflow language is very close to mathematical notation and can represent a wide class of relevant mathematical formulae. As a first step, SPIRAL attempts to deconstruct the original expression into simpler expressions, which, combined by a function composition, represent a data-flow graph of the computation [5]. The resulting expression is then translated into another language, called Σ -OL which adds the implicit representation of iterative computations. Next, the Σ -OL expression is rewritten using a series of rewrite rules, driven by the extensive knowledge base of SPIRAL’s optimization algorithms, into a shape which lends itself to generating the most efficient code for the target platform. Subsequently, a Σ -OL expression is compiled into an intermediate imperative language. By doing this, SPIRAL converts the dataflow graph into a sequence of loops and arithmetic operations. Finally, an intermediate imperative language representation, after some additional transformations, yields a C program which is compiled with an optimizing compiler, producing an executable high-performance machine code implementation of the original expression.

1.3 HELIX Introduction

When SPIRAL is applied to generate high-performance libraries used in mission critical software, the question arises as to what kind of assurances could be made about the correctness of the generated code. The goal of HELIX, as a part of the High Assurance SPIRAL project [6, 7], is to formally prove the correctness of SPIRAL optimizations and code generation using Coq proof assistant.

Similar to SPIRAL, HELIX works by transforming the original expression into a series of intermediate languages (as shown at Figure 1.2). The languages will be introduced in more detail in the following Section 3. The translation steps correspond to different levels of abstraction:

1. Mathematical formula
2. Dataflow (*HCOL*¹)
3. Dataflow with implicit loops (Σ -*HCOL*)
4. Dataflow with implicit loops, using memory blocks (*MHCOL*)
5. Imperative program (*DHCOL*)
6. Imperative program using machine numeric types (*floats* and *ints*) (*FHCOL*)
7. Mainstream low-level assembly language (LLVM IR)

Each language lowers the level of abstraction and introduces an additional level of detail, narrowing down the space of possible computational solutions of the given problem towards an exact algorithm. An expression is not only converted between the languages, but a series of transformations is performed at each language level. These transformations are optimization steps.

HELIX is not just a proof of SPIRAL correctness. It is inspired by SPIRAL and will typically be used in conjunction with SPIRAL but is different in several aspects:

- The projects have different primary goals. SPIRAL’s main objective is high-performance code generation, while HELIX focuses on high-assurance.

¹*HCOL* stands for Hybrid Control Operator Language.

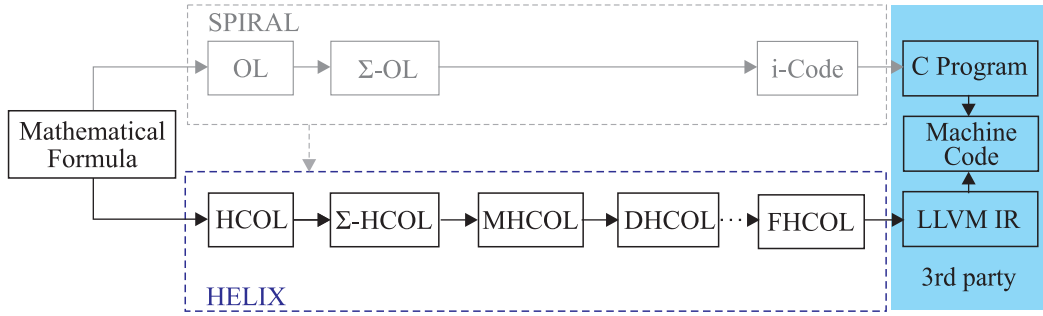


Figure 1.2: HELIX transformation stages

- There are two problems: 1) finding the optimal translation of a given formula to machine code for target architecture (search) and 2) verifying this translation. SPIRAL performs the former while HELIX is responsible for the latter. Currently, HELIX depends on SPIRAL as on a search oracle and just verifies SPIRAL results. In future, using HELIX as a foundation, some search functions could be transferred to it. This would open a door to formal verification of search/optimization algorithms instead of verifying their results.
- The final steps in the SPIRAL system are to generate a program in C language² and to compile it using a C compiler. In HELIX, we generate LLVM IR code, which verified compilation to machine code will be handled by 3rd party projects, like Vellvm. The main motivating factor in making this decision was the ease of formal verification; at the time, the only available formally verified C compiler (CompCert) lacked some necessary features (e.g. 64-bit support, vector instructions, intrinsics). However, we feel that LLVM IR is a better intermediate language for machine code generation. SPIRAL generates C code in a very restricted manner: it is in *static single assignment* (SSA)

²C is used when targeting standard general purpose ISA like x86 or ARM. SPIRAL also supports custom hardware platforms and GPUs where other languages like Verilog or CUDA are used instead of C.

form; only a few basic language constructs and operators are used; all loops have static bounds; compiler optimizations are disabled; and intrinsics are used to access low-level ISA capabilities, like vector instructions. Basically, it is used as a high-level assembly language. In our opinion, LLVM IR is better positioned to be used as such an assembly language. Additionally, the use of LLVM opens a few interesting possibilities, such as code generation for a wide list of hardware platforms via a growing list of LLVM backends and support for additional platform-specific optimization steps which could be verified using the same semantics framework. It must be noted that while the use of LLVM IR over C seems like a better approach architecturally, in practice, the maturity and support for modern C compilers at present makes C the more sensible choice, until the LLVM infrastructure matures further.

- SPIRAL is implemented in a heavily modified GAP [8] computer algebra system, while HELIX is implemented in Coq proof assistant.
- Originally, SPIRAL was built on the foundation of linear and multilinear operator theory. In early versions, all SPIRAL operators had to be multilinear. In later stages, SPIRAL added some support for non-linear operators [9]. HELIX, from the very beginning, was designed without the linearity assumption and supports non-linear operators.
- Due to the linear algebra lineage of SPIRAL, many of its core concepts were expressed using terminology and ideas from this field. HELIX, being a programming language (PL) project at its core, uses terminology and concepts from PL and type theories, formal methods, and functional programming. In other words, a computer science PL researcher will feel much at ease reading

HELIX papers and code while an algebraist will find the SPIRAL literature more tractable.

1.4 Motivating Example

Let us consider an application of HELIX to a real-life situation of high-assurance vehicle control [6] using a dynamic window vehicle control approach [10], as shown in Figure 1.3.

Given a physical model of vehicle dynamics, we generate a code to check a safety constraint. In this example, we will be checking whether it’s safe to proceed given the vehicle’s position, speed, and acceleration, with the location of an obstacle. The function will be invoked by the vehicle controller, and if it returns “false,” the vehicle will enter “fail-safe” mode, which would trigger emergency braking.

We will consider a ground robot driving on a flat, even surface. The robot is equipped with a distance measuring sensor (e.g. Lidar) which can detect and measure distance to obstacles. The obstacle sensor is sampled periodically, for example every 20ms and based on these measurements, the decisions on control outputs, such as steering, acceleration, and braking, are made. Once such decisions are applied to actuators controlling the robot, no further control changes can be made until the next sampling cycle. The obstacle does not have to stay static and can move.

This model allows the development of a control system which will help a robot avoid obstacles by stopping or steering around them. However, in this example, we will be considering only a *safety monitor* part of the system which ensures the *passive safety* property. Informally, that means there will be no collisions while the robot is driving, and collisions may occur only if the obstacle runs into the robot.

Without discussing detailed dynamic model of the system for which we refer interested readers to [6], we present the final formula for a dynamic window safety monitor in Equation (1.1). The soundness of the monitor formula from the point of view of cyber-physical control systems was proven in KeYmaera X [11].

$$\text{safe} \triangleq \|p_r - p_o\|_\infty > \frac{v_r^2}{2b} + V \frac{v_r}{b} + \left(\frac{A}{b} + 1\right) \left(\frac{A}{2}\epsilon^2 + \epsilon(v_r + V)\right) \quad (1.1)$$

The variables used in the safety monitor formula are:

- p_r - robot position
- p_o - obstacle position
- v_r - robot longitudinal speed
- V - maximum obstacle speed
- b - maximum braking (negative acceleration)
- A - maximum acceleration (positive)
- ϵ - sampling period

To complete the high-assurance chain for this monitor, we would like to have its implementation as machine code suitable for execution onboard, which is proven to be correct with respect to the mathematical formulation. Additionally, since this monitor will be executed in real-time and at high frequency, we would like the synthesized implementation to be efficient with respect to the target hardware.

We will use SPIRAL to generate high-performance implementation of the monitor function and verify it with HELIX. The first step is to express the input formula in SPIRAL *OL* and HELIX *HCOL* input languages. We will consider V , b , A and ϵ to be constants. The monitor function will take v_r , p_r , and p_o as inputs and return a Boolean value corresponding to the safety property. The data type of v_r is

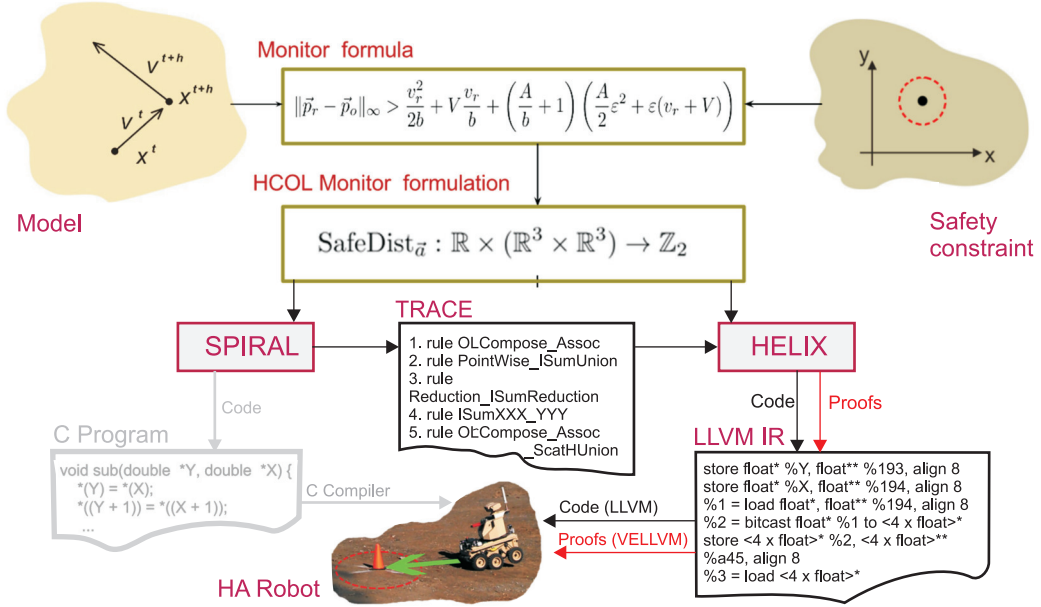


Figure 1.3: HELIX application to high-assurance vehicle control

scalar, and those of p_r and p_o are 2-dimensional coordinate vectors. Following [6], Equation (1.1) could be further rewritten as Equation (1.2), a form more suitable for *HCOL* representation.

$$\text{safe}_{V,A,b,\epsilon} : \mathbb{R} \times \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{Z}_2; \quad (1.2)$$

$$(v_r, p_r, p_o) \mapsto (p(v_r) < d_\infty(p_r, p_o))$$

$$d_\infty(\vec{x}, \vec{y}) \triangleq \|\vec{x} - \vec{y}\|_\infty \quad (1.3)$$

$$p(x) \triangleq a_2 x^2 + a_1 x + a_0 \quad (1.4)$$

$$a_0 = \left(\frac{A}{b} + 1 \right) \left(\frac{A}{2} \epsilon^2 + \epsilon V \right) \quad (1.5)$$

$$a_1 = \frac{V}{b} + \epsilon \left(\frac{A}{b} + 1 \right) \quad (1.6)$$

$$a_2 = \frac{1}{2b} \quad (1.7)$$

(1.8)

Running an *HCOL* expression representing (1.2) through SPIRAL will produce an unverified C-language implementation. During the processing, SPIRAL will invoke a number of sophisticated heuristics to find an optimal implementation for the target platform. The heuristics result in application of a sequence of semantically preserving transformation steps to the original expression to reshape it into optimized form. The list of transformation steps will be recorded in a “trace” file produced by SPIRAL. The C code will depend on target platform parameters. For the simple case where the target platform is set to generic x86 without SIMD instructions, the C implementation of the dynamic monitor is shown in Listing B.2.

Parameters of the generated function are:

D[0] - a_0
D[1] - a_1
D[2] - a_2
X[0] - v_r
X[1] - $p_r.x$ (x component of the vector p_r)
X[2] - $p_r.y$ (y component of the vector p_r)
X[3] - $p_o.x$ (x component of the vector p_o)
X[4] - $p_o.y$ (y component of the vector p_o)

The return values are 1 for **True**, 0 for **False**.

For comparison, in Listing B.3, we show the C code, which is generated for an x86 platform with Intel Streaming SIMD Extensions (SSE) enabled. This version includes runtime floating point error analysis using the *online uncertainty propagation* approach, described in Section 5.3.2. In addition to 1 for **True** or 0

`False` false, this version could also return -1 , which means that the safety property could not be reliably computed for the given input values.

HELIX will start with the same input formula. Unlike SPIRAL, it does not make decisions about what transformation steps to perform, but rather it relies on SPIRAL’s trace for the list of steps to perform. However, in HELIX, each of these steps is backed up by a formally proven lemma which guarantees that it preserves the semantics of the expression. HELIX will re-create SPIRAL’s transformation of the expression, but this time, each step, and as a result, the whole sequence of transformations, will be formally proven to be correct. In the unlikely event that SPIRAL would suggest a non-semantically preserving transformation, HELIX will reject this step and report an error.

Finally, the optimized expression will be compiled by HELIX into LLVM IR assembly language (as shown in Listing B.1). This compilation will also be proven correct, and the generated code will be guaranteed to correctly implement the computations described by the input expression, up to some error bounds of numerical accuracy related to limitations due to use of floating point calculations instead of real numbers in the original formula.

The LLVM IR program could be further compiled by the LLVM compiler toolchain, into machine code for the target hardware platform. At this time, there are no formally verified IR compiler backends, but they are included in the roadmap of the DeepSpec project [12].

As shown by this example, HELIX provides a framework of automated translation from a mathematical formula to an efficient and formally verified implementation in machine code.

This formulation of the dynamic window monitor was used in High Assurance

SPIRAL [6] in the scope of the DARPA HACMS project. The target vehicles were the LandShark robot by Black-I Robotics³ and a simulated car modeled after an American built car, shown in Figure 1.4.



Figure 1.4: HACMS vehicles

³<http://www.blackirobotics.com/>

Chapter 2

Background

In this section, we provide a brief history and accessible introduction of formal verification and the key concepts of formal semantics of programming languages. The fields of constructive mathematics, type theory, and formal verification are vast, so we will limit our scope to the theories, developments, tools, and approaches relevant to our work.

2.1 Formal Verification

A *formal verification* of a computer program involves formally proving that it satisfies some set of properties, often collectively referred to as *formal specification*. The specification does not deal only with the expected results of computation in a purely mathematical sense, but it can also state requirements on computational complexity, use of memory and resources, program inputs and outputs, etc.

The result of such verification is a formal proof of the compliance of the program to the given specification. Such proof is usually required to be machine-checkable because, for most practical applications, the proofs are too big to be

checked by hand. Most proofs make some assumptions about program inputs or the execution environment. These assumptions must be clearly stated as part of the specification. It is important to understand that the formal guarantees provided will be constrained by these assumptions.

2.2 Mathematical Foundations

The idea of formal verification of computer programs is rooted in constructive mathematics and proof theory. Although there were earlier attempts, the idea of formalization of mathematics was most fully explored by Alfred North Whitehead and Bertrand Russell in their seminal work, *Principia Mathematica* [13]. Most modern formalizations of mathematics are based on Zermelo-Fraenkel set theory (ZF) with or without the *axiom of choice*.

The idea of *constructive mathematics* is that to prove the existence of a mathematical object, one needs to be able either to produce it or describe a way to construct it from other objects. While there is still discussion about whether constructivism is the right approach for building a comprehensive foundations of all mathematics, it has been shown to be very useful and practical for describing a substantial part of the body of modern mathematics. Constructive proofs have been shown for the Fundamental Theorem of Algebra [14, 15], the Fundamental Theorem of Integral Calculus [16], and others. There are several varieties of constructive mathematics. One variety of particular interest is Per Martin-Löf's Constructive Type Theory [17], where he developed the notion of *dependent types*, which allowed Thierry Coquand to develop his Calculus of Constructions [18], a variant of which, Calculus of Inductive Constructions (CIC), is used today for the constructive foundation of mathematics used in Coq [19] proof assistant.

The concept of dependent types is fairly straightforward yet very powerful. In a simple, non-dependent type system, every term a has a simple type A , usually written as $\mathbf{a}:A$. Dependent types are, in essence, type-valued functions [20] that send terms to types. Thus, we can have a term b which has type T parameterized by a term c , written as $\mathbf{b}: T(c)$. One example of a dependent type frequently used in HELIX is the fixed-size vector type. A naïve way to represent vectors is to use unbounded lists. The disadvantage of this approach is that for an operation which is only defined for vectors of certain size (e.g. *dot product*), this constraint can not be enforced at compilation time by the type signature and must be checked at runtime. The dependent type of vector of length n with elements of type A would be `vector (A:Type) (n:N)`. Using this type, we can define a dot product as `dot: vector A n → vector A n → A`, and Coq typechecker will ensure that it can be applied only to vectors of size n .

The last founding piece is the Curry-Howard Isomorphism [21], which could be described informally as establishing a correspondence between constructive proofs and computable programs and between logical propositions and types. Thus, proving a logical implication $A \supset B$ is the same as writing a function in typed lambda calculus with type $A \rightarrow B$.

2.3 Coq Proof Assistant

The historic developments described in the previous section paved the way to build the Coq proof assistant we use in this work¹. The mathematical theorems are expressed in a dependent type system. The proofs are constructed either automatically (using *tactics*) or handwritten but verified by the proof assistants' small *trusted kernel*.

¹Many other proof assistants exist. Isabelle, Lean, Idris, Nuplr, and Agda to name a few.

nel, which interprets them as programs in the Calculus of Inductive Constructions.

With Coq you can:

- Define functions and predicates.
- State mathematical theorems and software specifications.
- Interactively develop formal proofs of theorems.
- Machine-check these proofs.
- Extract Coq programs to languages like OCaml, Haskell, or Scheme.

Coq architecture [22] is shown in Figure 2.1. Coq consistency could not be proven in Coq itself because this would contradict Gödel’s second incompleteness theorem. Thus, Coq contains a small trusted code base, written by hand but not formally verified. Whenever we trust a proof in Coq, we implicitly trust that this bit of the code is correct. The trusted code base consists of three main parts. The *parsing* part is responsible for parsing input programs in Coq’s input languages: Gallina and *Ltac*. The *extraction* part is used only when Coq’s programs are “extracted” into languages like Haskell or OCaml. Finally, the *kernel* type-checks the Calculus of Inductive constructions. This feature of having a small independent kernel which can check proof terms by relatively small and comprehensible algorithms is called the *De Bruijn criterion* for proof assistants [23]. Coq’s kernel is relatively small (18 KLOC) and has been carefully written and reviewed². Still, critical bugs are found in it at an approximate rate of one per year[22]. The rest of the implementation (the untrusted code) does not need to be trusted, as the results of its processing are ultimately passed through the trusted kernel, which will either accept or reject it if incorrect. Whenever a tactic is invoked, it must produce a *proof term*, which will be type-checked by the kernel, and the proof will be accepted only if it passes the check. For example, one can develop Coq’s extension which will introduce a

²<https://coq.discourse.group/t/coq-trusted-kernel-code-size/1012/2>

new tactic command, which will claim to automatically prove a class of unprovable goals. However, this tactic can not produce a correct proof term which would pass the kernel check. This safety check will prevent proofs from using this erroneous tactic.

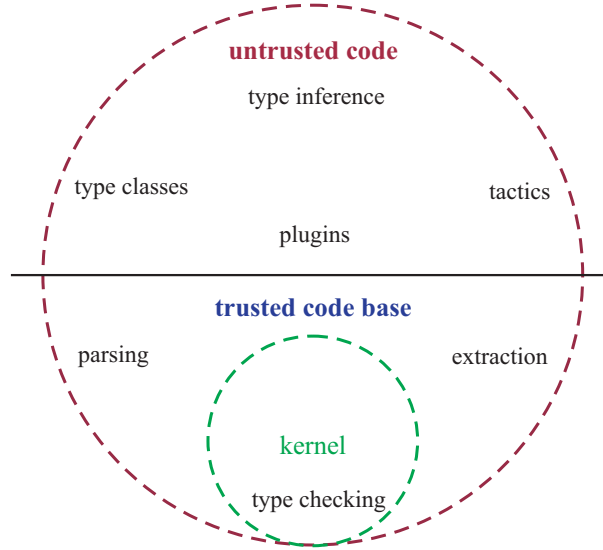


Figure 2.1: Coq architecture

Coq has demonstrated its usefulness for reasoning about a variety of problems. Coq’s applications include:

- *Properties of programming languages.* The *CompCert* certified compiler project [24], the *Bedrock* verified low-level programming library [25].
- *Formalization of mathematics.* The full formalization of the *Feit-Thompson theorem* [26], *Fundamental theorem of calculus* [16], *Four color theorem* [27], and *Homotopy type theory* [28].
- *Verification of distributed protocols.* Blockchain consensus algorithms [29].
- *Verification of cryptography.* Encryption and digital signature schemes [30], zero-

knowledge protocols [31], and hash functions [32].

2.3.1 Quick Introduction to Coq

The two most important categories of objects Coq distinguishes between are those belonging to the **Prop** sort and those belonging to the **Type** sort. The first is used for logical propositions, while the second is used for arbitrary data types. Types can be defined inductively. For example, the natural number type has two constructors: **0** (zero) and **S** (successor function):

```
Inductive N :=  
  | 0 : N  
  | S : N → N.
```

This definition allows us to express any natural number. For example, zero is just **0**, one is **(S 0)**, two is **(S (S 0))**, etc. Recursive functions can be defined using the **Fixpoint** command. Since we know that by construction any term of type **N** is either **0** or **S n** for some $n \in \mathbb{N}$, we can use *pattern-matching* on its constructors to define natural number addition:

```
Fixpoint plus (n m : N) : N :=  
  match n with  
  | 0 ⇒ m  
  | S p ⇒ S (plus p m)  
end
```

Coq only accepts definitions of functions that are guaranteed to terminate. This limitation is needed to ensure logical consistency of the system, as well as the decidability of the type-checking. In our **plus** definition, we recurse on a direct subterm of n . In such case, Coq is able to automatically infer a termination proof. In other cases, the termination must be proven by the user.

Finally, we can state and prove theorems about the objects we define. Let $+$ be notation for `plus`. Now, we can prove a simple arithmetic theorem, stating that $2 + 3 = 5$:

```
Theorem plus_2_3 : (S (S 0)) + (S (S (S 0))) = (S (S (S (S (S 0))))).
Proof.
  simpl. (* expand definition of plus *)
  reflexivity. (* use definition of equality *)
Qed.
```

We construct a proof using *tactics*. In the example above, `simpl` is a tactic that performs basic application of definitions, and `reflexivity` proves equality between two syntactically equal terms (modulo some reductions).

Another statement we can prove is $\forall n, 0 + n = n$:

```
Theorem plus_0_n : (∀ n, 0 + n = n).
Proof.
  intros n. (* take any n *)
  simpl. (* expand definition of plus *)
  reflexivity. (* use definition of equality *)
Qed.
```

To prove $\forall n, n + 0 = n$, a slightly different version of the previous statement, we need to use a *mathematical induction*:

```
Theorem plus_n_0 : (∀ n, n + 0 = n).
Proof.
  (* proof by induction on n *)
  induction n; cbn.
  (* base case *)
  - reflexivity.
  (* inductive step *)
  - rewrite IHn. (* apply induction hypothesis *)
```

```

    reflexivity.
Qed.

```

By Curry-Howard Isomorphism, the proof of this statement corresponds to the following functional program:

```

plus_n_0 = λ n : ℕ ⇒
    nat_ind (λ n0 : ℕ ⇒ n0 + 0 = n0) eq_refl
    (λ (n0 : ℕ) (IHn : n0 + 0 = n0) ⇒
        eq_ind_r (λ n1 : ℕ ⇒ S n1 = S n0) eq_refl IHn) n
: ∀ n : ℕ, n + 0 = n

```

where `nat_ind` is a term with a type that corresponds to the natural number induction principle, and `eq_refl` is a term of type $\forall x, x = x$.

2.4 Formal Semantics of Programming Languages

In this section, we introduce a few core concepts from formal semantics of programming languages. This is not a comprehensive introduction into the topic and [33] should be a better reference.

A programming language is defined by its *syntax* describing how the program looks and *semantics* describing how it behaves. A program matching the formal syntax definition is said to be *well-formed*. For mainstream programming languages, like Java or Python, we can further distinguish between *concrete syntax* and *abstract syntax*. The former deals with the representation of a program as a sequence of characters as stored on disk or shown in a text editor. Concrete syntax is concerned with minutiae of spacing, indentation, separators, quoting, keywords, etc. The concrete syntax is normally processed by a compiler or an interpreter's *parsing pass* into an internal representation in memory, typically an *abstract syntax tree* (AST)

which should comply to the language’s *abstract syntax*.

Formal verification of concrete syntax parsing by hand is laborious and rarely done. A more practical approach is to use a parser generator which generates parser implementation based on formal syntax definition in Backus-Naur Form (BNF) style. Then, such a parser generator could be either proven to be correct or extended to automatically generate proof that the generated parsing code indeed complies to the syntax description [34, 35, 36].

An alternative approach often used in domain-specific languages (DSLs) is to skip the concrete syntax definition and to “embed” the *target language* in some other existing *host language*. There are two flavors of doing this: a *shallow* and a *deep* embedding. In the case of shallow embedding, the target language is defined as a subset of the host language constructs, inheriting both their syntax and semantics. In case of deep embedding, the syntax of the target language (the AST) is encoded as a data structure in the host language, and its semantics are provided separately by additional definitions in the host language. There are also combinations of these two approaches, which are generally referred to as “mixed embedding.”

An AST is just a data structure, and the abstract syntax is to ensure that this structure represents a well-formed program. Depending on the expressiveness of the host language, the abstract syntax compliance may or may not need additional checks. For example, when using such a powerful language as Gallina with its inductive and dependent types, it is often possible to enforce an AST compliance to an abstract syntax in AST’s data type definition, making all ASTs represent only well-formed programs “by construction.”

There are several approaches for ascribing a formal semantics to a programming language. We are interested in the following ones, as they are relevant to this

work:

Big-step operational semantics is also sometimes referred to as *evaluation semantics* or *natural semantics*. It defines what would be the result of a completed execution of a program and is usually implemented as an interpreter written in the host language. For imperative languages, such an evaluation function maps the initial state of memory and environment to the final state after program execution.

ITrees-based semantics is a novel approach to defining the semantics of programming languages in terms of *interaction trees* [37]. This approach offers modular, compositional, and executable semantics [38] and can represent impure, potentially non-terminating, and mutually recursive computations.

2.5 Certified Compilation and Translation Validation

Given a pair of languages with formally defined syntax and semantics, the question is how to develop a compiler which translates between them while providing some formal guarantees about compilation results.

We need to start with the question of what guarantees we want to provide. On the highest level, we usually want to guarantee the *semantic preservation property*. Informally, that states that the semantics of a program in the source language remain the same after compilation to the target language. Usually the two semantics are compared in terms of *observable behaviors*, such as whether the program diverges or terminates, how it modifies memory, and whether the traces of system calls match [24].

The two main approaches relevant to our work are:

Compiler verification requires proving that the given compiler for all well-formed

programs in the source language always produces programs in the target language which are semantically equivalent to the originals. This is a high-level description, but in real-life implementations, one also needs to consider invalid programs, programs too complex for a compiler to handle, and programs with undefined behaviors. See [24] for further discussion on how these could be handled.

Translation validation does not require proving that the compiler is always correct. Instead, given a program in the source language and a corresponding program in the target language, presumably produced by the compiler, it requires proving their semantic equivalence. This is done by writing a *validator* which either returns a proof of such equivalence or fails if it can find none. Obviously, this is a more limited approach than the full compiler verification since in the worst case, it requires manually writing a validator for each compiled program. However, a generic validator which checks semantics preservation for a pair of programs could be written with prior knowledge of the structure of code that the target compiler produces, so the validation proofs can be automated. Ideally, such automation would validate all programs produced by a given compiler automatically.

Chapter 3

HELIX Formalization

As discussed in Section 2.4, in order to reason about the properties of programs in the languages used in HELIX, we first need to formally define their syntax and semantics in a form which would allow us to reason about them.

HELIX is implemented in Coq with all its languages embedded in Gallina. All translation steps are also implemented in Coq or Template-Coq [39] using several techniques which are discussed in detail in Section 4.

In this section, we present all HELIX languages, shown in Figure 1.2. A quick summary of the languages is shown in Table 3.1. We chose to embed all HELIX languages in Coq Proof assistant [19], which allowed us to reason about them in Coq’s Calculus of Inductive Constructions using its tactics language.

As can be seen from the table, the type representing numerical data differs among the languages. We start from \mathcal{R} , which abstracts \mathbb{R} in *HCOL*, and end up with IEEE floating-point numbers in LLVM IR. Similarly, vector data representation also evolves. We start with dense finite-size vectors as the main data type of *HCOL* which, in the final translation, are represented as memory blocks referenced

Name	Scalars	Vectors	Embedding	Type	Error Handling
<i>HCOL</i>	\mathcal{R}	dense vector	shallow	declarative	no
Σ - <i>HCOL</i>	\mathcal{R}_θ	sparse vector	mixed	functional	no
<i>MHCOL</i>	\mathcal{R}	memory blocks	mixed	functional	yes
<i>DHCOL</i>	\mathcal{R}	env. + memory	deep	imperative	yes
<i>FHCOL</i>	IEEE double	env. + memory	deep	imperative	yes
LLVM IR	IEEE double	env. + memory	deep	imperative	yes

Table 3.1: Summary of HELIX languages

by variables in LLVM IR. Although all languages are embedded in Gallina, different types of embedding are used, as shown in Table 3.1. Also, as we proceed down the translation chain, we transition from purely functional to imperative languages. Finally, while programs in our input language *HCOL* are always “correct by construction” (as ensured by Coq’s type system), once such a program is transitioned into an imperative form later in the chain, the error handling is introduced. The details of all these implementation aspects will be discussed for each HELIX language in detail in the following sections.

Aside from general principle of lowering the abstraction level of each step, the choice of languages was driven by several considerations:

- The first two languages, *HCOL* and Σ -*HCOL*, were designed by formalization of SPIRAL *OL* and Σ -*OL* languages. They have to match these languages modulo syntax to allow us to use SPIRAL results in the translation validation approach.
- The last language in HELIX is LLVM IR. Our first choice was to use C language, as SPIRAL does. However, at the time, the only certified compiler available, CompCert, did not support some of the features we needed (e.g. 64-bit, vector instructions). Also, the C language was overkill for the code

generation we wanted to perform. We chose LLVM IR, a lower-level language which allows us to express more directly the shape of the machine code we ultimately want to produce.

- The decision to use LLVM IR instead of C also affected our choice of the previous language in the chain. Both in SPIRAL and HELIX, the last intermediate language is a high-level imperative language which is capable of expressing the algorithms we synthesize but is high-level enough to allow reasoning and proofs about these algorithms without going into lower-level details (like memory and pointers) of languages like C or IR. SPIRAL uses `i-Code` language for this purpose which is quite close to C. Since in HELIX, we no longer use C, we have more flexibility in defining this intermediate imperative language. The language we devised, *DHCOL*, is indeed imperative but of a higher level than `i-Code`, more specialized, and better shaped for formal reasoning and translation to IR. For example, it uses de Bruijn indices for variables, has logically scoped memory allocation, and shares the memory model with Vellvm’s IR semantics.
- The split between *DHCOL* and *FHCOL* is motivated by our desire to clearly define the boundary between abstract numeric types (*DHCOL*) and concrete machine types (*FHCOL*). To simplify numeric analysis, we want these languages to be very similar, representing the same sequence of computation steps but for different types.
- The *MHCOL* language was added for pure convenience to bridge the gap between functional Σ -*HCOL* and the imperative *DHCOL*. Much of the proof for this transition has to do with representing Σ -*HCOL* vectors in memory.

To simplify these proofs, we found it convenient to introduce memory block abstraction first, while keeping the language functional.

All HELIX languages are embedded in Coq but use different types of embedding. We chose a *shallow embedding* for *HCOL* as it reflects its algebraic nature in the most natural way. Σ -*HCOL* uses functional abstraction, which also fits well with the shallow embedding (as Gallina is a functional language). It is especially convenient to represent the operator families used in iterative operators as functions. However, because of sparsity, Σ -*HCOL* operators need to carry some additional information. In particular, the *sparsity contract*. Thus, we chose the *mixed embedding* where each operator is a record which contains a shallow embedded operator implementation along with two sets representing the sparsity contract. This form still allows us to use the *setoid rewriting* technique. *MHCOL* representation is similar to that of Σ -*HCOL*. To express additional logical properties of operators for all these languages, we used *typeclasses*. Finally, for *DHCOL*, we switched to *deep embedding*. This allowed us to clearly separate the language abstract syntax from its semantics and, in particular, to assign more than one semantics to the language.

3.1 *HCOL* Language

HELIX *HCOL* language is based on the SPIRAL *OL* language which was originally designed to represent linear algebra expressions on real or complex vectors. The primitive *HCOL* operators are functions from vectors to vectors. Higher-order operators, such as function composition, allow the building of more complex *HCOL* expressions.

Since HELIX uses SPIRAL as an oracle, the *HCOL* and *OL* languages must be compatible. *HCOL* is a formalization of *OL*, deep embedded in Coq. Any well-

formed *OL* expression could be trivially mechanically translated into the corresponding *HCOL* expression. Unlike *OL*, *HCOL* expressions are always well-formed as we use Coq’s powerful dependent type system to enforce this. Thus, for example, vector dimensions will always match when constructing complex *HCOL* expressions from elementary operators. Similarly, constants and arithmetic expressions representing indices of vector elements will be properly bound.

By varying the dimensions of vectors, *HCOL* could represent computations with different levels of granularity. The structure of such expressions represents the dataflow graph of computation. By applying a set of rewriting rules, an *HCOL* expression could be gradually “broken down,” synthesizing the dataflow to match the hardware architecture of the target system.

HCOL is a shallow-embedded language in Coq proof assistant. All *HCOL* operators are represented as functions in Coq’s host language, *Gallina*. The limitation of this approach is that operators must be *total functions* which can be proven to terminate. This poses no problem in practice, as all *HCOL* operators we encountered fit this definition. The following are the data types used in *HCOL*:

The *Carrier Type*: Unlike *OL*, the main data type is abstract instead of using concrete types, such as \mathbb{R} or \mathbb{C} . *HCOL* \mathcal{R} is an abstract representation of such a numeric type, expressed in terms of its algebraic properties. See description below in Section 3.1.1 for details.

Finite-dimensional Vectors: To represent vectors, we use the inductively-defined `Vector` type from Coq’s standard library. Vector elements have type \mathcal{R} . We will use Coq notation `avector n` or mathematical notation \mathcal{R}^n interchangeably to describe vectors of \mathcal{R} of length n .

Finite natural numbers: Some HELIX operators use finite natural numbers as

vector offsets. They are upper-bounded to ensure not to exceed the expected target vector size. They are encoded using Coq’s `sig` type to represent a number along with the proof that it has passed the range check. In this paper, we sometimes use the shorthand notation \mathbb{I}_n to denote $\{x : \mathbb{N} \mid x < n\}$ type.

The dimensions of input and output vectors of an *HCOL* operator are encoded as indices of the `vector` type family, and vector type \mathcal{R}^n corresponds to `(vector \mathcal{R} n)` in Coq. When constructing a complex *HCOL* expression, Coq’s type system ensures that the dimensions of all components match.

3.1.1 Carrier Type

This is an abstract representation of a numeric type, expressed in terms of its algebraic properties. Definitions and proofs formulated for the carrier type can be applied, for example, to \mathbb{R} , \mathbb{Q} , or \mathbb{Z} , as they satisfy these properties.

We denote the carrier type as \mathcal{R} . Algebraic properties are expressed using corresponding typeclass instances from the *MathClasses* library [40]. It is postulated that there are instances for \mathcal{R} of the following typeclasses:

- Equality: `Equiv`, `Setoid` (see discussion below in Section 3.1.2.).
- Constants: `Zero`, `One`.
- Operators: `Plus`, `Mult`, `Negate`, `Abs`.
- Abstract algebra structures: `Ring`.
- Comparisons: `Lt`, `Le`.
- Decidability: `Decision (x=y)`, `Decision (x<y)`.
- Ordering: `TotalOrder`, `StrictSetoidOrder`, `SemiRingOrder`, `FullPseudoOrder`.

For example, we require that \mathcal{R} , along with the corresponding operations,

forms an algebraic ring, has a total ordering, and has decidable *equality*. Some operators do not require all these properties, but to be able to construct homogeneous *HCOL* expressions, the carrier type imposes the superposition of all the constraints required by all *HCOL* operators. Additionally, it is assumed that the \mathcal{R} type is populated with some special values, like **zero** and **one**.

All these properties are assumed, by admitting instances of the corresponding typeclasses. This provides an axiomatic system of abstract \mathcal{R} .

3.1.2 Equality

The definition of equality is essential for *HCOL* operator rewriting. The Coq default notion of equality (**eq**) is too restrictive for our purposes. For example, it would not allow us to work with rational numbers represented by non-reduced integer fractions. Depending on what concrete type is used in place of the abstract carrier type, we would like to be able to define a meaningful equality relation for this type. In general, we would like to work on a carrier type equipped with an equivalence relation, which is also called a *setoid*.

Operational typeclass **Equiv** defines an **equiv** relation for a given type. its subclass, **Setoid**, additionally requires this relation to be an *equivalence relation* by presenting proofs that it is transitive, commutative, and reflexive.

Since we have declared our carrier type \mathcal{R} to be an instance of a **Setoid** typeclass, we can define **equiv** for vectors of this type as a *pointwise relation* which makes them also a setoid: $\forall (n:\mathbb{N}), \text{Setoid } (\text{vector } \mathcal{R} \ n)$.

From that follows the natural definition of the *HCOL* operator *extensional equality* which states that two operators F and G are equal if for all possible input vectors x , the values of $(F x)$ and $(G x)$ are also equal.

As we work mostly with setoid equality instead of Coq’s standard equality, we follow the MathClasses library convention of using the $(=)$ notation to refer to the `equiv` relation instead of Coq’s standard `eq`. To refer to standard Coq’s `eq` equality, we use the notation (\equiv) . These notations are followed throughout this thesis.

3.1.3 *HCOL* operators

Our current formalization of *HCOL* includes the following operators, shallow-embedded in Coq.

HPointwise: $\forall (n \in \mathbb{N}), (\mathbb{I}_n \rightarrow \mathcal{R} \rightarrow \mathcal{R}) \rightarrow \mathcal{R}^n \rightarrow \mathcal{R}^n$

This operator applies a given function to each element of the input vector, as shown in Figure 3.1. The function takes two arguments: the element’s index and its value. The output is the vector of the same length as the input vector.

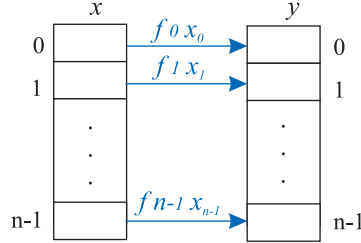


Figure 3.1: HPointwise operator

Let us look more closely at the definition of the HPointwise operator in Coq:

```
Definition HPointwise {n: ℕ} (f: ℐn → ℛ → ℛ) (x: vector ℛ n) : vector ℛ n
:= Vbuild (λ j jd ⇒ f (mkFinNat jd) (Vnth x jd)).
```

Listing 3.1: HPointwise operator definition

The operator is implemented using the `Vbuild` function from the *CoLoR* library [41]. The definition is fairly straightforward; `HPointwise` generates a vector of length n where an element with index j is the result of the application of the j -th function from family f to the input vector x .

$$\underline{\underline{\text{HAtomic}: (\mathcal{R} \rightarrow \mathcal{R}) \rightarrow \mathcal{R}^1 \rightarrow \mathcal{R}^1}}$$

This operator “lifts” a scalar valued function to an *HCOL* operator on single element vectors.

$$\underline{\underline{\text{HScalarProd}: \forall (n \in \mathbb{N}), \mathcal{R}^{n+n} \rightarrow \mathcal{R}^1}}$$

Calculates the dot product of two vectors. The input vectors are concatenated and passed as a single vector of size $n + n$. The result is returned as a single-element vector. For an input vector $x = [x_0, x_1, \dots, x_{n+n-1}]$ it computes $[x_0 \cdot x_n + x_1 \cdot x_{n+1} + \dots + x_{n-1} \cdot x_{n+n-1}]$.

$$\underline{\underline{\text{HBinOp}: \forall (n \in \mathbb{N}), (\mathbb{I}_n \rightarrow \mathcal{R} \rightarrow \mathcal{R} \rightarrow \mathcal{R}) \rightarrow \mathcal{R}^{n+n} \rightarrow \mathcal{R}^n}}$$

This operator applies a given function to pairs of elements from two halves of an input vector, as shown in Figure 3.2. The function additionally takes an index in the range of 0 to $n - 1$ as an argument.

$$\underline{\underline{\text{HReduction}: \forall (n \in \mathbb{N}), (\mathcal{R} \rightarrow \mathcal{R} \rightarrow \mathcal{R}) \rightarrow \mathcal{R} \rightarrow \mathcal{R}^n \rightarrow \mathcal{R}^1}}$$

This operator performs a *right fold* of a vector. The two parameters are a binary function and the initial value. The result is returned as a vector size 1.

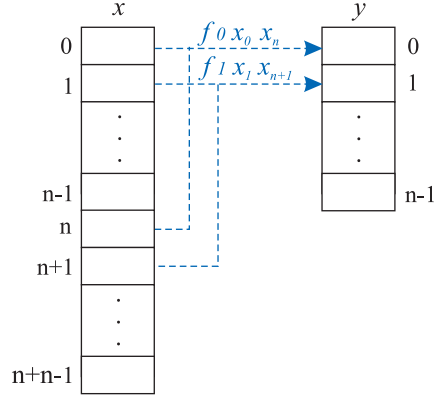


Figure 3.2: HBinOp operator

HEvalPolynomial: $\forall (n \in \mathbb{N}), \mathcal{R}^n \rightarrow \mathcal{R}^1 \rightarrow \mathcal{R}^1$

This operator computes a univariate polynomial of an n -th degree. It is parameterized by a vector of constant coefficients. The input and output scalar values are represented as vectors of size 1.

For input $\vec{x} = [x_0]$ and parameter $\vec{a} = [a_0, a_1, \dots, a_n]$, it computes $[a_0 + a_1 \cdot x_0 + a_2 \cdot x_0^2 + \dots + a_n \cdot x_0^n]$.

HPrepend: $\forall (m, n \in \mathbb{N}), \mathcal{R}^m \rightarrow \mathcal{R}^n \rightarrow \mathcal{R}^{m+n}$

This operator concatenates the input vector of size n with the constant vector “prefix” of size m .

HMonomialEnumerator: $\forall (n \in \mathbb{N}), \mathcal{R}^1 \rightarrow \mathcal{R}^{n+1}$

This operator computes a list of positive integer powers (0 to n) of a single variable. Assuming the input is a single-element vector $\vec{x} = [x_0]$, it returns $[1, x_0, x_0^2, \dots, x_0^{n-1}]$.

HInductor: $\forall (n \in \mathbb{N}), (\mathcal{R} \rightarrow \mathcal{R} \rightarrow \mathcal{R}) \rightarrow \mathcal{R} \rightarrow \mathcal{R}^1 \rightarrow \mathcal{R}^1$

This operator is a *recursor* [42] applied up to the depth n . The parameters are a binary function f and the initial value z . For example, if the input is a single-element vector $\vec{x} = [x_0]$, for $n = 5$, it computes $[f (f (f (f (f z x_0) x_0) x_0) x_0) x_0]$. It is analogous to **Fold** operator in *Wolfram Mathematica* [43].

HInduction: $\forall (n \in \mathbb{N}), (\mathcal{R} \rightarrow \mathcal{R} \rightarrow \mathcal{R}) \rightarrow \mathcal{R} \rightarrow \mathcal{R}^1 \rightarrow \mathcal{R}^n$

This operator is similar to **HInductor** but returns all intermediate values for depths up to n . For example, if the input is a single-element vector $\vec{x} = [x_0]$, for $n = 3$, it computes $[z, f z x_0, f (f z x_0) x_0]$. It is analogous to **FoldList** operator in *Wolfram Mathematica*.

HInfinityNorm: $\forall (n \in \mathbb{N}), \mathcal{R}^n \rightarrow \mathcal{R}^1$

This operator computes an *infinity norm* $\|x\|_\infty$ of a given input vector and returns it as a single-element vector.

HChebyshevDistance: $\forall (n \in \mathbb{N}), \mathcal{R}^{n+n} \rightarrow \mathcal{R}^1$

This operator computes the *Chebyshev distance* between two vectors of size n . The vectors are concatenated and passed as a single input vector of size $n + n$. The resulting scalar value is returned as a single-element vector.

HVMinus: $\forall (n \in \mathbb{N}), \mathcal{R}^{n+n} \rightarrow \mathcal{R}^n$

This operator represents subtraction of two vectors of size n . The vectors are concatenated and passed as an input vector of size $n + n$.

$$\underline{\underline{\text{HCross: } \forall (n_1, n_2, m_1, m_2 \in \mathbb{N}), (\mathcal{R}^{m_1} \rightarrow \mathcal{R}^{n_1}) \rightarrow (\mathcal{R}^{m_2} \rightarrow \mathcal{R}^{n_2}) \rightarrow (\mathcal{R}^{m_1+m_2} \rightarrow \mathcal{R}^{n_1+n_2})}}$$

This is a higher-order operator implementing the *cartesian product* of two operators, sometimes called a *tupling combinator*. When applied to operators f and g , it produces a new operator which takes as input a pair of vectors (x_0, x_1) (concatenated) and returns $(f\ x_0, g\ x_1)$ (also concatenated).

$$\underline{\underline{\text{HStack: } \forall (n_1, n_2, m \in \mathbb{N}), (\mathcal{R}^m \rightarrow \mathcal{R}^{n_1}) \rightarrow (\mathcal{R}^m \rightarrow \mathcal{R}^{n_2}) \rightarrow (\mathcal{R}^m \rightarrow \mathcal{R}^{n_1+n_2})}}$$

This is a higher-order operator implementing parallel application of two operators. When applied to operators f and g it produces a new operator which for input vector x and returns $(f\ x, g\ x)$ (concatenated).

$$\underline{\underline{\text{HCompose: } \forall (m, n, t \in \mathbb{N}), (\mathcal{R}^t \rightarrow \mathcal{R}^n) \rightarrow (\mathcal{R}^m \rightarrow \mathcal{R}^t) \rightarrow (\mathcal{R}^m \rightarrow \mathcal{R}^n)}}$$

This is a higher-order operator for operator composition. When applied to operators f and g it produces a new operator $(f \circ g)$.

$$\underline{\underline{\text{HTLess: } \forall (n, m_1, m_2 \in \mathbb{N}), (\mathcal{R}^{m_1} \rightarrow \mathcal{R}^n) \rightarrow (\mathcal{R}^{m_2} \rightarrow \mathcal{R}^n) \rightarrow (\mathcal{R}^{m_1+m_2} \rightarrow \mathcal{R}^n)}}$$

This is a higher-order operator implementing element-wise comparison of the results of the cartesian product of two operators. When applied to the operators f and g , it produces a new operator which takes as input a pair of vectors (x_0, x_1) (concatenated) and returns a vector produced by element-wise “less than” comparison of $f\ x_0$ and $g\ x_1$ using a decidable `lt` predicate, which must be defined for \mathcal{R} . If the predicate is `True`, the corresponding output vector’s element will be `zero` or `one` otherwise. The `zero` and `one` are special values in \mathcal{R} .

3.1.4 Example

An *HCOL* formulation of the dynamic window monitor expression (1.2) introduced in Section 1.4 is shown in Listing 3.2.

```
Definition dynwin_orig (a: avector 3): avector (1 + (2 + 2)) → avector 1
:= HTLess (HEvalPolynomial a) (HChebyshevDistance 2).
```

Listing 3.2: Dynamic Window Monitor in *HCOL*

3.2 Σ -*HCOL* Language

Most vector and matrix operations can be expressed as iterative computations on their elements. To generate machine code for such computations, we transform our expressions into a form where these iterations become explicit.

The goal of our next language, Σ -*HCOL* is to represent algebraically iterative computations on vectors. Where *HCOL* operates on whole vectors, Σ -*HCOL* allows for finer granularity introducing operations on individual elements.

An iterative computation on vectors can be viewed as superposition of computations performed during each step which processes only a subset of elements. The vector positions not used during an iteration step can be left undefined. This can be represented naturally with sparse vectors. For example, an element-wise function application to elements of a dense vector can be represented as the sum of columns of a diagonal sparse matrix, as shown in Figure 3.3. In this example, for simplicity, we use \mathcal{R}^n type to represent sparse real-valued vectors of length n and assume that sparse cells hold a special *structural zero* value, which is treated as regular zero under addition. Later in this section, we will give a more formal treatment of how we represent and reason about sparsity in HELIX.

$$\text{Map}_f \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} f(a_0) \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ f(a_1) \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ f(a_2) \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ f(a_3) \end{bmatrix} = \begin{bmatrix} f(a_0) \\ f(a_1) \\ f(a_2) \\ f(a_3) \end{bmatrix}$$

Figure 3.3: Dense vector as sum of four sparse vectors

Assuming that f is implemented in C as `void f(\mathcal{R} *src, \mathcal{R} *dst)`, it roughly corresponds to the following loop:

```
for (int i=0; i<4; i++)
{
    f(src+i, dst+i);
}
```

which requires four iterations. If we have a vectorized implementation of f^2 with type $f^2 : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ which is implemented in C as `void f2(\mathcal{R} src[2], \mathcal{R} dst[2])`, the sum would look like Figure 3.4.

$$\text{Map}_{f^2} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} f^2(a_0, a_1) \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ f^2(a_2, a_3) \\ 0 \end{bmatrix} = \begin{bmatrix} f(a_0) \\ f(a_1) \\ f(a_2) \\ f(a_3) \end{bmatrix}$$

Figure 3.4: Dense vector as sum of two sparse vectors

It roughly corresponds to the following loop, which now requires only two iterations:

```
for (int i=0; i<2; i++)
{
    f2(src+2*i, dst+2*i);
}
```

In these examples, f can be viewed as an abstraction for a scalar CPU instruction, such as x86 *FADD*, and $f2$ can be a SIMD version of it, similar to *ADDPS* x86 SSE instruction.

In essence, sparsity allows us to represent partial computations. For instance, in Figure 3.3, we use an operation (e.g. addition) and the default value for the sparse elements (e.g. zero), which form a *monoid*, to represent superposition of partial computations algebraically. Maintaining algebraic abstraction allows us to transform and to prove equality of operations on vectors, representing various computation flows.

3.2.1 Modelling Missing Values and Collisions

In our formalization, each sparse vector’s element could be either an actual value or a *structural* value. One can think about a structural value as an empty cell with a default placeholder value assigned to it.

To ensure proper factorization of a complex computation into superposition of elementary ones, we need to make sure that the calculation of each vector’s element is assigned to exactly one elementary computation. That means that when combining vectors representing the results of two computations, we should never combine two non-empty vector elements. When combining vector elements pairwise, one of the values must be structural. If both values are non-structural, we call this a *collision*, which indicates there are two conflicting paths trying to perform computation of the same value. Any collision detected should be tracked down the computation tree, and any operation with a value produced as a result of that collision should be marked as colliding as well. Normally, the well-formed Σ -*HCOL* expression triggers no collisions (see Section 3.2.5 on how we ensure this).

The first, naïve approach to tracking structural and collision flags is to use a product type which contains the actual value and two flags: $\mathcal{R} \times \mathbb{B} \times \mathbb{B}$. However in many situations, we only care about actual values and want to avoid dealing with structural flags implicitly. Our solution is to use *Writer Monad* to track the structural properties of carrier type \mathcal{R} , as described below.

```
Record  $\mathcal{R}_{\text{flags}}$  : Type := mk $\mathcal{R}_{\text{flags}}$  {is_struct:  $\mathbb{B}$ ; is_collision:  $\mathbb{B}$ }.
```

In the snippet above, we first define a record data type $\mathcal{R}_{\text{flags}}$ which holds structural and collision flags. To use these flags in Writer Monad (using *ExtLib* [44] library), we also need to define a *Monoid* record:

```
Definition mzero := mkRthetaFlags  $\top \perp$ .
Definition mappend (a b:  $\mathcal{R}_{\text{flags}}$ ) :  $\mathcal{R}_{\text{flags}}$  :=
  mkRthetaFlags
    (is_struct a && is_struct b)
    (is_collision a || is_collision b ||
      (negb (is_struct a || is_struct b))).
Definition Monoid_ $\mathcal{R}_{\text{flags}}$  : Monoid  $\mathcal{R}_{\text{flags}}$  := Build_Monoid mappend mzero.
```

Listing 3.3: Exclusive Union Monoid

The initial flags' value called *mempty* has the structural flag *true* and the collision flag *false*. The *mappend* operation above is used to combine the two sets of flags. It works as follows. If one of the operands is non-structural, the result is also non-structural. The collision flags are "sticky;" once set for either operand, they are propagated into the result. Finally, attempting to combine two non-structural elements raises a new collision.

It could be shown that the following Monoid laws are satisfied:

$$\forall a, \quad a \oplus 0 = 0 \oplus a = a \tag{3.1}$$

$$\forall a, \forall b, \forall c, \quad (a \oplus b) \oplus c = a \oplus (b \oplus c) \quad (3.2)$$

Here, we use 0 for *mempty* and \oplus for *mappend*. In Coq, we just declare an instance of the *MonoidLaws* typeclass for our newly defined Monoid record and prove properties from the Equations (3.1) and (3.2).

```
Instance MonoidLaws_ℛ_flags: MonoidLaws Monoid_ℛ_flags.
```

To track the flags while performing operations on the values of type \mathcal{R} , we use Writer Monad, parametrized by a Monoid which defines how flags will be handled:

```
Variable fm:Monoid ℛ_flags.
Definition Monad_ℛ_flags := writer fm.
Definition ℛ_fm := Monad_ℛ_flags ℛ.
```

To construct values of the type \mathcal{R}_{fm} , we define two convenience functions:

```
Variable fm:Monoid ℛ_flags.
Definition mkStruct (v:ℛ) : ℛ_fm fm := ret v.
Definition mkValue (v:ℛ) : ℛ_fm fm := tell (mkℛ_flags ⊥ ⊥) ;; ret v.
```

For most Σ -*HCOL* operators, we are interested in the $\mathcal{R}_{\text{flags}}$ type parametrized by a Monoid with *exclusive union* as a *mappend* operation, as shown in Listing 3.3. We will call this type \mathcal{R}_θ . A commonly used constant θ of this type holds \mathcal{R} ring's *additive identity* (0) as a value and has the structural flag set and the collision flag unset:

```
Definition ℛ_θ := ℛ_fm Monoid_ℛ_flags.
Definition θ : ℛ_θ := mkStruct 0.
```

To illustrate how Writer Monad is used to track the flags, let us examine how they are combined when a binary operation is performed on underlying values. Recall the *execWriter* will return the flags value of type $\mathcal{R}_{\text{flags}}$ for a given monadic

value of type \mathcal{R}_θ . To apply the binary operation `op` to the underlying \mathcal{R} values, we use *liftM2* to promote it to a monad. Now, just by unfolding the underlying definitions, it could be trivially shown that:

```

∀ (op:  $\mathcal{R} \rightarrow \mathcal{R} \rightarrow \mathcal{R}$ ) (a b:  $\mathcal{R}_\theta$ ),
  execWriter (liftM2 op a b) = mappend (execWriter a) (execWriter b).

```

In other words, that arguments' flags will be combined using the *mappend* operation. Similarly, using `evalWriter` to unwrap the writer monad to extract the underlying value, it could be shown that lifting a binary operation will result in a value computed by applying it to the unwrapped arguments:

```

∀ (op:  $\mathcal{R} \rightarrow \mathcal{R} \rightarrow \mathcal{R}$ ) (a b:  $\mathcal{R}_\theta$ ),
  evalWriter (liftM2 op a b) = op (evalWriter a) (evalWriter b).

```

The \mathcal{R}_θ is not the only parametrisation of \mathcal{R}_{fm} we use. Other monoids could be used to provide different strategies for combining flags. For example, another useful monoid uses the same *mzero* value as in Listing 3.3 except with the *mappend*' function which tracks the preexisting collisions without generating new ones:

```

Definition mappend' (a b:  $\mathcal{R}_{\text{flags}}$ ) :  $\mathcal{R}_{\text{flags}}$  :=
  mkRthetaFlags
    (is_struct a && is_struct b)
    (is_collision a || is_collision b).

Definition Monoid_ $\mathcal{R}_{\text{flags}}$ ' : Monoid  $\mathcal{R}_{\text{flags}}$  :=
  Build_Monoid mappend' mzero.

```

Listing 3.4: “Safe” Union Monoid

This monoid instance allows us to define a “safe” variant of the \mathcal{R}_θ type as $\mathcal{R}_{\theta'} \triangleq \mathcal{R}_{\text{fm}} \text{ Monoid_}\mathcal{R}_{\text{flags}}'$. This type could be used, for example, in scenarios where iteration does not represent partial computations. To mix these different types of

iterations, a conversion between \mathcal{R}_θ and $\mathcal{R}_{\theta'}$ is defined, which preserves both flags and values (see **SafeCast** and **UnsafeCast** operators below).

Some definitions do not depend on the monoid used to specialize \mathcal{R}_{fm} , and in the rest of this section, we use type $\mathcal{R}_- \triangleq \mathcal{R}_{\text{fm}} \text{ fm}$ and assume that all equations using this type are universally quantified over $\text{fm} \in \text{Monoid } \mathcal{R}_{\text{flags}}$.

Frequently, we need to combine vectors element-wise using the provided binary scalar function $\text{dot} : \mathcal{R} \rightarrow \mathcal{R} \rightarrow \mathcal{R}$. For each output vector element, the values are computed applying dot to the corresponding elements of the two input vectors. The flags are combined using the **mappend** operation from the provided monoid. This operation is called **Vec2Union**:

Definition **Vec2Union** $\{\text{n}:\mathbb{N}\}$ ($\text{dot} : \mathcal{R} \rightarrow \mathcal{R} \rightarrow \mathcal{R}$)
 $: \text{svector fm n} \rightarrow \text{svector fm n} \rightarrow \text{svector fm n}.$

Listing 3.5: **Vec2Union** vector combining operation

When $\text{fm} = \text{Monoid_}\mathcal{R}_{\text{flags}}$, this operation represents the combination of two partial computations. On the other hand, when $\text{fm} = \text{Monoid_}\mathcal{R}_{\text{flags}'}$, it is just an element-wise combination of the two sparse vectors using the provided binary operation. For example, if $\text{dot} = \text{plus}$, it is just vector addition.

3.2.2 Index Mapping Functions

Sometimes, we will use functions to express the relationship between the indices of two vectors. We call such functions *index mapping functions*.

An index mapping function f has a domain of natural numbers \mathbb{N} in interval $[0, m)$ (denoted as \mathbb{I}_m) and the range of \mathbb{N} in interval $[0, n)$ (denoted as \mathbb{I}_n and encoded as $\{x : \mathbb{N} \mid x < n\}$ type in Coq).

$$f^{m \rightarrow n} : \mathbb{I}_m \rightarrow \mathbb{I}_n$$

Such a function, for example, could be used to establish a relation between the indices of two vectors with respective sizes m and n .

3.2.3 Families of Index Mapping Functions

We can extend our notion of an index mapping function into a *family of index mapping functions*. We define a *family* f of k index mapping functions:

$$\forall j < k, \quad f_j^{m \rightarrow n} : \mathbb{I}_m \rightarrow \mathbb{I}_n \quad (3.3)$$

Such families could be used, for example, to represent the individual index maps used per loop iteration. The non-collision property corresponds to the *injectivity* of a family of index mapping functions, and the totality of computation corresponds to *bijectivity*.

The family is called *injective* if it satisfies:

$$\forall a, \forall b, \forall i, \forall j, \quad f_a(i) = f_b(j) \implies (i = j) \wedge (a = b) \quad (3.4)$$

The family is called *surjective* if it satisfies:

$$\forall j, \exists a, \exists i, f_a(i) = j \quad (3.5)$$

The family is called *bijective* if it is both *injective* and *surjective*.

The subscript indices in the mathematical notation above are just additional

arguments, and the actual type of the function is:

$$f : \mathbb{I}_k \rightarrow \mathbb{I}_m \rightarrow \mathbb{I}_n \quad (3.6)$$

Or in *uncurried* form:

$$f^{m \rightarrow n} : (\mathbb{I}_k \times \mathbb{I}_m) \rightarrow \mathbb{I}_n \quad (3.7)$$

In which case, the standard definitions of surjectivity and injectivity apply.

3.2.4 Operator Type

Σ -*HCOL* operators are defined using mixed embedding. By that, we mean that an operator's implementation is Gallina function from vectors to vectors, which is wrapped up in a record that holds some additional information. The full definition is shown in Listing 3.6.

```
Record SHOperator {i o: ℕ} {svalue: CarrierA} {fm: Monoid RthetaFlags} : Type :=
mkSHOperator {
  op: svector fm i → svector fm o ;
  op_proper: Proper ((=) ==> (=)) op;
  in_index_set: FinNatSet i ;
  out_index_set: FinNatSet o;
  svalue_at_sparse: ∀ v,
    (∀ j (jc:j<o), ¬out_index_set (mkFinNat jc) → evalWriter (Vnth (op v) jc) = svalue);
}.
```

Listing 3.6: SHOperator type

The operator record type is indexed by:

i,o Input and output vector dimensions. Vector sizes are static and must match when building complex expressions from elementary operators.

svalue The default value which will be used to initialize new sparse cells.

fm Flags monoid instance. It defines how sparsity flags will be handled.

The fields of the `SHOperator` record are:

op Functional, shallow-embedded, implementation of the operator.

op_proper Proper morphism instance for *op* function.

in_index_set, out_index_set Sparsity patterns for input and output vectors, encoded as sets of finite natural numbers with bounds corresponding to dimensions of input and output vectors, respectively.

svalue_at_sparse The guarantee (proof) that the operator's output will contain the *svalue* at sparse indices.

It should be noted that this definition of an operator provides no guarantees that the implementation will respect sparsity patterns. This will be ensured via *structural properties*, discussed next.

3.2.5 Structural Correctness

Expressions must be in a certain shape which lends itself to efficient code generation. Ensuring such a shape is a problem distinct from semantics preservation, and we have defined a separate set of properties to ensure what we call “structural correctness.” It involves reasoning about the underlying operations performed on sparse vectors using a monad to track sparsity and detect structural errors.

Each operator definition includes two sets, `in_index_set` and `out_index_set`, representing its *sparsity contract*. They define the expected sparsity patterns of input vectors and the guaranteed sparsity patterns of output vectors.

We have also defined the following structural properties which guarantee that a Σ -*HCOL* expression is in a form which is suitable for optimal and correct code generation:

1. The sparsity contract (`in_index_set` and `out_index_set` membership) is decidable.
2. Only the values at indices from the `in_index_set` of the input vector affect the output.
3. During operator evaluation, a sufficiently filled input vector (values at all indices in the `in_index_set`) guarantees a properly filled output vector (values at all indices in the `out_index_set`).
4. An operator evaluation will never generate values at indices which are not present in the `out_index_set`.
5. As long as there are no collisions at indices in the `in_index_set` in the input vector, none will be produced at indices in the `out_index_set` in the output vector.
6. An operator evaluation will never generate collisions at indices outside the `out_index_set` of the output vector.

We have grouped these properties in a `SHOperatorFacts` type class and have proven its instances for all Σ -*HCOL* operators that we have defined. The proof of these properties for higher-order operators is *compositional*; as long as all operators involved are instances of `SHOperatorFacts`, it can be shown that all Σ -*HCOL* higher-order operators are also instances of `SHOperatorFacts`. That gives us a structural correctness proof “by construction” for any Σ -*HCOL* expression.

3.2.6 Operator Families

Similar to families of index mapping functions, we can have families of operators.

We define a *family* F of k operators as:

$$\forall \text{fm}, \forall i, \forall o, \forall \text{svalue}, \quad F : \mathbb{I}_k \rightarrow \text{SHOperator fm } i \text{ o svalue} \quad (3.8)$$

All operators in the family use the same monoid, the same input and output dimensions, and the same default structural value.

3.2.7 Equality

For Σ -*HCOL*, we need to define the notion of equality for scalar values, vectors, and operators, as we did for *HCOL* in Section 3.1.2. Here again, we use `Equiv` typeclass to define our equality relation for various types as described below.

For scalar values of type $(\mathcal{R}_{\text{fm}} \text{ fm})$, the equality relation is defined for any monoid $\text{fm} \in \text{Monoid } \mathcal{R}_{\text{flags}}$. It is defined as an equality of the underlying values of type \mathcal{R} :

```
Instance  $\mathcal{R}_{\text{fm\_equiv}}$ : Equiv  $(\mathcal{R}_{\text{fm}} \text{ fm})$  :=  
   $\lambda \text{ am bm} \Rightarrow (\text{evalWriter am}) = (\text{evalWriter bm})$ .
```

For sparse vectors of this type, we use pointwise equality.

Finally for Σ -*HCOL* operators, the equality is defined as extentional equality of the underlying shallow-embedded implementations:

```
Instance  $\text{SHOperator\_equiv } \{i \text{ o} : \mathbb{N}\} \{svalue : \mathcal{R}\}$ :  
  Equiv  $(\text{@SHOperator } i \text{ o svalue})$  :=  $\lambda \text{ a b} \Rightarrow \text{op a} = \text{op b}$ .
```

At first glance, the definition is missing the comparison of sparsity patterns. It can be shown that, as defined, the relation is strong enough to guarantee that

the sparsity patterns will also match, assuming both the operators are *structurally correct*.

3.2.8 Σ -*HCOL* Operators

There are fourteen Σ -*HCOL* operators described below. Unless specified otherwise, they operate on sparse vectors with elements of type $(\mathcal{R}_{\text{fm}} \text{ fm})$ for any $\text{fm} \in \text{Monoid } \mathcal{R}_{\text{flags}}$.

Embed

```
Embed (svalue:  $\mathcal{R}$ ) (n b:  $\mathbb{N}$ ) (bc:  $b < n$ ) : @SHOperator fm 1 n svalue.
```

Takes an element from a single-element input vector and puts it at a specific index in a sparse vector of given length.

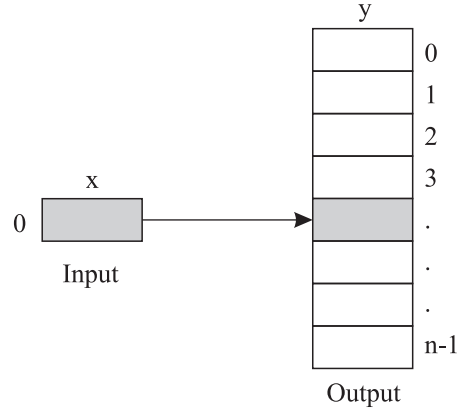


Figure 3.5: Embed dataflow

Pick

```
Pick (svalue:  $\mathcal{R}$ ) (n b:  $\mathbb{N}$ ) (bc:  $b < n$ ) : @SHOperator fm n 1 svalue.
```

Selects an element from the input vector at the given index and returns it as a single element vector.

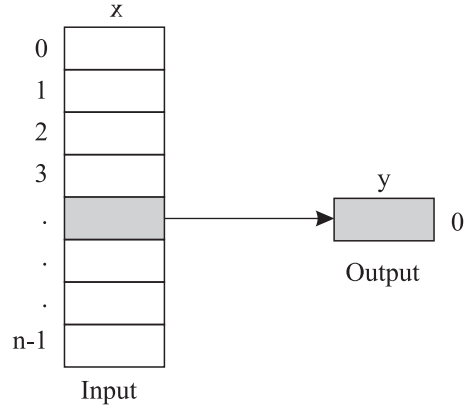


Figure 3.6: Pick dataflow

Scatter

```
Scatter (svalue:  $\mathcal{R}$ ) (n m:  $\mathbb{N}$ ) (f: index_map n m) {f_inj: index_map_injective f} :
  @SHOoperator fm n m svalue.
```

Embedding can be generalized where more than one element can be embedded at once. The destination selection is controlled by a user-provided *index mapping function*.

The operator maps elements of the input vector to the elements of the output according to an index mapping function f . The mapping is *injective* but not necessarily *surjective*. That means the output vector could be sparse.

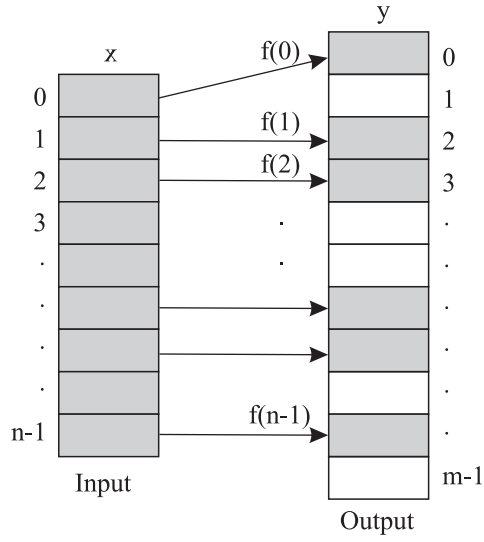


Figure 3.7: Scatter dataflow

Gather

```
Gather (svalue:  $\mathcal{R}$ ) (n m:  $\mathbb{N}$ ) (f: index_map m n) :
  @SHOperator fm n m svalue.
```

Picking can be generalized where more than one element can be picked at once. The element selection is controlled by a user-provided *index mapping function*.

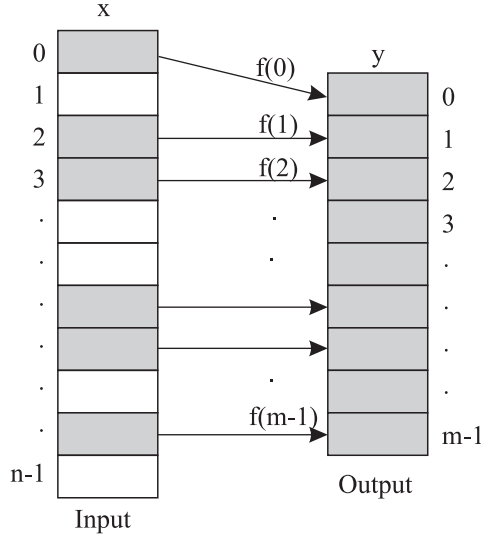


Figure 3.8: Gather dataflow

liftM_HOperator

```
liftM_HOperator (svalue:  $\mathcal{R}$ ) (i o:  $\mathbb{N}$ )
  (op:  $\text{avector } i \rightarrow \text{avector } o$ )
  '{HOP: HOperator i o op}
  : @SHOperator fm i o svalue.
```

This operator allows “lifting” *HCOL* operators, so they can be used in Σ -*HCOL* expressions. Since *HCOL* operates on dense vectors, the input vector is first “densified” by dropping structural flags and replacing sparse values with the *svalue*. During this operation, structural values become indistinguishable from non-structural values. After applying the *HCOL* operator, the result is “sparsified” (converted to sparse vector format) by marking all values as non-structural.

SHPointwise

```
SHPointwise (svalue:  $\mathcal{R}$ ) (n:  $\mathbb{N}$ )
```

```

(f: FinNat n →  $\mathcal{R}$  →  $\mathcal{R}$ )
‘{pF: !Proper ((=) ⇒ (=) ⇒ (=)) f}
: @SHOperator fm n n svalue

```

This is a Σ -*HCOL* version of the `HPointwise` operator from *HCOL* (see Section 3.1.3). We could not just lift `HPointwise` via `liftM_Hoperator` because we want to preserve structural flags. However, it can be shown that the two implementations are equal with respect to `SHOperator_equiv` which only compares values and ignores flags.

SHBinOp

```

SHBinOp (svalue:  $\mathcal{R}$ ) (n:  $\mathbb{N}$ )
  (f: FinNat n →  $\mathcal{R}$  →  $\mathcal{R}$ )
  ‘{pF: !Proper ((=) ⇒ (=) ⇒ (=)) f}
  : @SHOperator fm (n+n) n svalue.

```

This is a Σ -*HCOL* version of the `HBinOp` operator from *HCOL* (see Section 3.1.3). Just like with `SHPointwise`, can not just lift the `HBinOp` via the `liftM_Hoperator` because we want to preserve structural flags. However, it can be shown that the two implementations are equal with respect to `SHOperator_equiv` which only compares values and ignores flags.

SHInductor

```

SHInductor (svalue:  $\mathcal{R}$ ) (n:  $\mathbb{N}$ )
  (f: FinNat n →  $\mathcal{R}$  →  $\mathcal{R}$ )
  ‘{pF: !Proper ((=) ⇒ (=) ⇒ (=)) f}
  (initial:  $\mathcal{R}$ )
  : @SHOperator fm 1 1 svalue.

```

This is a Σ -*HCOL* version of the **HInductor** operator from *HCOL* (see Section 3.1.3). The initial value is treated as *non-structural*.

Apply2Union

```

Apply2Union (i o:  $\mathbb{N}$ )
  (svalue:  $\mathcal{R}$ )
  (dot:  $\mathcal{R} \rightarrow \mathcal{R} \rightarrow \mathcal{R}$ )
  '{dot_mor: !Proper ((=)  $\implies$  (=)  $\implies$  (=)) dot}
  '{scompat: BFixpoint svalue dot}
  (f g: @SHOperator fm i o svalue)
  : @SHOperator fm i o svalue.

```

This is a higher-order operator applying two operators to the same input and combining their results using **Vec2Union** (See Listing 3.5).

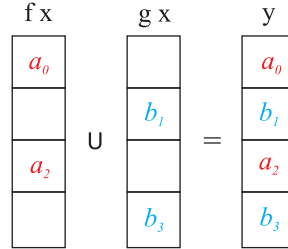


Figure 3.9: Apply2Union dataflow

The additional constraint is that the default value for sparse cells (**svalue**) is a fixpoint of the provided binary function **dot**, defined as **dot svalue svalue = svalue**. This is to assure that the value of sparse cells is preserved when combining them.

SafeCast

```

SafeCast {svalue:  $\mathcal{R}$ } {i o:  $\mathbb{N}$ }
  (f: @SHOperator Monoid_ $\mathcal{R}_{\text{flags}}$ ' i o svalue)
  : @SHOperator Monoid_ $\mathcal{R}_{\text{flags}}$  i o svalue.

```

This is a higher-order operator wrapping another Σ -*HCOL* operator. While it does not change the values computed by the wrapped operator, it swaps the monadic wrapper used to track sparsity properties from $\text{Monoid}_{\mathcal{R}_{\text{flags}}}'$ to $\text{Monoid}_{\mathcal{R}_{\text{flags}}}$. As we recall from Section 3.2.1, the former does not track collisions, while the latter does.

UnsafeCast

```

UnsafeCast {svalue:  $\mathcal{R}$ } {i o:  $\mathbb{N}$ }
  (f: @SHOperator Monoid_ $\mathcal{R}_{\text{flags}}$  i o svalue)
  : @SHOperator Monoid_ $\mathcal{R}_{\text{flags}}'$  i o svalue.

```

Similar to **SafeCast** but with wrappers switched in the opposite direction, from $\text{Monoid}_{\mathcal{R}_{\text{flags}}}$ to $\text{Monoid}_{\mathcal{R}_{\text{flags}}}'$.

SHCompose

```

SHCompose {svalue:  $\mathcal{R}$ }
  {i1 o2 o3:  $\mathbb{N}$ }
  (op1: @SHOperator o2 o3 svalue)
  (op2: @SHOperator i1 o2 svalue)
  : @SHOperator i1 o3 svalue.

```

This performs a functional composition of operators. The **op2** is applied to the input first. The result is then used as an input of **op1**. Sometimes, we use shortcut notation ($\text{op1} \odot \text{op2}$) alluding to the function composition operator \circ .

IReduction

```
IReduction {svalue:  $\mathcal{R}$ } {i o n:  $\mathbb{N}$ }  
  (dot:  $\mathcal{R} \rightarrow \mathcal{R} \rightarrow \mathcal{R}$ )  
  '{pdot: !Proper ((=)  $\implies$  (=)  $\implies$  (=)) dot}  
  '{scompat: BFixpoint svalue dot}  
  (op_family: @SHOperatorFamily Monoid_ $\mathcal{R}_{\text{flags}}$ ' i o n svalue)  
  : @SHOperator Monoid_ $\mathcal{R}_{\text{flags}}$ ' i o svalue.
```

This iteratively applies an indexed family of n operators to the input and combines their outputs element-wise using the provided binary function `dot`. See additional discussion on implementation of this operator below in Section 3.2.10.

The additional constraint is that the default value for sparse cells (`svalue`) is a fixpoint of provided binary function `dot`, defined as `dot svalue svalue = svalue`. This is to assure that the value of sparse cells is preserved when combining two of them.

This operator is defined for vectors of $\mathcal{R}_{\theta'}$ as its intended use is to computationally combine the results of a family of operators, and it must not treat combining non-sparse values as errors (collisions).

IUnion

```
Definition IUnion {svalue:  $\mathcal{R}$ } {i o n:  $\mathbb{N}$ }  
  (dot:  $\mathcal{R} \rightarrow \mathcal{R} \rightarrow \mathcal{R}$ )  
  '{pdot: !Proper ((=)  $\implies$  (=)  $\implies$  (=)) dot}  
  '{scompat: BFixpoint svalue dot}  
  (op_family: @SHOperatorFamily Monoid_ $\mathcal{R}_{\text{flags}}$  i o n svalue)  
  : @SHOperator Monoid_ $\mathcal{R}_{\text{flags}}$  i o svalue :=
```

Iteratively applies an indexed family of n operators to the input and combines

their outputs element-wise using the provided binary function `dot`. See additional discussion on implementation of this operator below.

The additional constraint is that the default value for sparse cells (`svalue`) is a fixpoint of the provided binary function `dot`, defined as `dot svalue svalue = svalue`. This is to assure that the value of sparse cells is preserved when combining them.

This operator is defined for vectors of \mathcal{R}_θ and represents an abstraction for a parallel loop, combining the results of partial computations. To allow *loop-level parallelism* in a well-formed Σ -*HCOL* expression, partial computations should never overlap. Such overlapping can not occur as long as the sparsity patterns of `op_family` members are disjoint. The collision tracking built-in in \mathcal{R}_θ allows us to prove this.

3.2.9 Sparse Embedding

One class of *HCOL* expressions that we are particularly interested in has the following form:

$$\text{SparseEmbedding}_{f,g,K} \triangleq (\lambda i. \text{Scat}_{f_i} \circ K_i \circ \text{Gath}_{g_i}) \quad (3.9)$$

The parameters are:

- A family of k index mapping function $g^{m \rightarrow t}$
- A family of k “kernel” operators K
- An *injective* family of k index mapping function $f^{\ell \rightarrow n}$

This form is called a *sparse embedding* of an operator family K (the *kernel*) and could be used as a step in iterative processing of a vector’s elements. It

corresponds to the body of a loop with k iterations in which the *gather* picks the input vector's elements, which are then processed by K , and the results are then dispatched to appropriate positions in the output vector using *scatter*. The index function family f must be injective. The **SparseEmbedding** is monoid-agnostic and defined for vectors of \mathcal{R}_- .

This is a very flexible and powerful construct. We can process vector elements one by one or in groups. The order of processing is controlled by index mapping functions. It allows us to model various memory access patterns useful for SIMD or CPU cache related optimizations.

3.2.10 Map-Reduce

The **IUnion** and **IReduction** *HCOL* operators are variants of the same operation, which we will call *map-reduce*. The higher-order *map-reduce* operation $\mathbf{MR}_{k,f,z}$ takes an indexed family of k operators (typically a *sparse embedding*) and produces a new operator. It has the following type:

$$\mathbf{MR}_{k,f,z}: (\mathbb{N} \rightarrow (\mathcal{R}_-^n \rightarrow \mathcal{R}_-^m)) \rightarrow \mathcal{R}_-^n \rightarrow \mathcal{R}_-^m \quad (3.10)$$

When evaluated, *map-reduce* applies all family members with indices between 0 and $k - 1$ (inclusive) to an input vector, and the resulting k vectors are folded element-wise using a binary function ($f : \mathcal{R} \rightarrow \mathcal{R} \rightarrow \mathcal{R}$). The initial value ($z : \mathcal{R}_-$) is used in the first folding step and treated as a *structural* value.

A simple example applies a function f to all elements of a vector of size 2:

$$\mathbf{MR}_{2,+,0}(\lambda i. (\mathbf{Scat}_{\lambda x.i} \circ \llbracket f \rrbracket \circ \mathbf{Gath}_{\lambda x.i})) \quad (3.11)$$

When using *map-reduce* in **IReduction**, the results of family members' applications must be dense. In the case of **IUnion**, the body of *map-reduce* should be a family of *sparse embeddings*. The dataflow of expression (3.11) is shown in Figure 3.10.

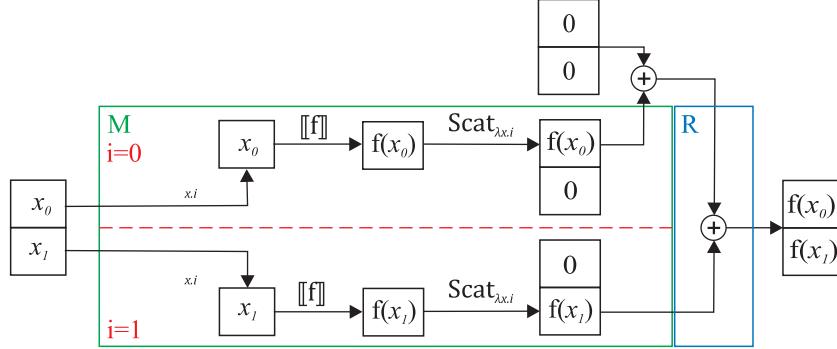


Figure 3.10: Map-Reduce of a Sparse Embedding

3.2.11 Relation between *HCOL* and Σ -*HCOL*

Lifting *HCOL* operators to be used in Σ -*HCOL* allows a temporary mixture of abstractions, corresponding to embedding mathematical formulae in a functional program. A Σ -*HCOL* expression can be gradually transferred to a purely functional form by applying a series of rewriting rules.

In iterative factorization of operations on vectors, each iteration represents a partial computation which outputs a sparse vector. In SPIRAL, the sparsity is represented by default values (typically 0) assigned to sparse cells. No tracking is performed which historically leads to many difficult to find implementation bugs. In Σ -*HCOL*, we have implicit sparsity tracking and a special sparse vector type. Thus, while in SPIRAL, Σ -*OL* is a superset of *OL*, in HELIX, Σ -*HCOL* and *HCOL* are two distinct languages operating on different data types: sparse vs. dense vectors.

Σ -*HCOL* operators represent partial computations and are defined on sparse vectors, unlike *HCOL* operators which represent total computations and are defined on dense vectors.

3.2.12 Example

A result of the initial translation of an *HCOL* formulation of the dynamic window monitor from Listing 3.2 to Σ -*HCOL* is shown in Listing 3.7. It has been abridged to hide some non-essential parameters.

```

Definition dynwin_SHCOL (a: avector 3): @SHOperator Monoid_ℛflags (1+(2+2)) 1 zero :=
  SafeCast (SHBinOp (IgnoreIndex2 Zless))
  ⊙ Apply2Union plus
    (ScatH 0 1
      ⊙ (liftM_HOperator (@HReduction plus 0)
        ⊙ SafeCast (SHBinOp (IgnoreIndex2 mult))
        ⊙ liftM_HOperator (HPrepend a )
        ⊙ liftM_HOperator (HInduction 3 mult one))
      ⊙ GathH 0 1)
    (ScatH 1 1
      ⊙ liftM_HOperator (@HReduction max 0)
      ⊙ SHPointwise (IgnoreIndex abs)
      ⊙ (SumSparseEmbedding (n:=2)
        (λ jf ⇒ SafeCast
          (SHBinOp (o:=1)
            (Fin1SwapIndex2 jf (IgnoreIndex2 sub))))
        (λ j ⇒ h_index_map (proj1_sig j) 1)
        (λ j ⇒ h_index_map (proj1_sig j) 2))
      ⊙ GathH 1 1).

```

Listing 3.7: Dynamic Window Monitor in Σ -*HCOL*

3.3 *MHCOL* Language

MHCOL is an intermediate step in the HELIX transformation chain between purely functional Σ -*HCOL* and imperative *DHCOL* languages. Imperative language seman-

tics, as we see in later sections, ultimately describes how program execution steps update the memory state. Sparse vectors in Σ -*HCOL* are an algebraic abstraction for *memory blocks*. We make this explicit in the intermediate mixed-embedded language, *MHCOL* (M stands for *memory*). While *MHCOL* is still a functional language, we bring it closer to the next language transformation step by changing data representation from vectors to memory blocks. Each memory block is represented as a finite map from memory offsets to values of a carrier type. There is no mappings for keys corresponding to sparse values. Besides physical data representation, the main change from Σ -*HCOL* is that we made the sparsity implicit. Starting from *MHCOL*, we no longer the maintain algebraic abstraction which we used to transform and optimize Σ -*HCOL* expressions. There are no “default” values for sparse cells. That means that reading a sparse element is an error which leads to introduction of implicit error handling in *MHCOL*.

An example of both representations is shown in Figure 3.11. It shows a sparse vector with three initialized cells, A, B, and C, and one sparse cell with default value 0. The memory representation of the same vector uses a dictionary with three elements. There is no mapping for key 1 corresponding to vector’s sparse cell.

0	A		
1	0	0	\rightarrow A
2	B	2	\rightarrow B
3	C	3	\rightarrow C

Sparse vector
Dictionary

Figure 3.11: Sparse vectors as dictionaries

Generally speaking, there is a 1-to-1 correspondence between Σ -*HCOL* and *MHCOL* operators with the main difference in the input and output data types.

MHCOL uses memory blocks, where Σ -*HCOL* uses sparse vectors. An example application of the **MApply2Union** operator in *MHCOL* is shown in Figure 3.12. It is similar to the **Apply2Union** Σ -*HCOL* operator in Figure 3.9 but operates on memory blocks instead of vectors. Each of the two operators **f** and **g** are applied to the input memory block **x** producing corresponding dictionaries with disjoint keys $\{0, 2\}$ and $\{1, 3\}$, respectively. They are then merged into the final resulting dictionary **y**. Unlike Σ -*HCOL*, merging memory blocks does not involve combining cells with matching keys using a binary operation. The memory blocks are simply merged. If there is a value associated with a key in both input blocks, it is considered an error.

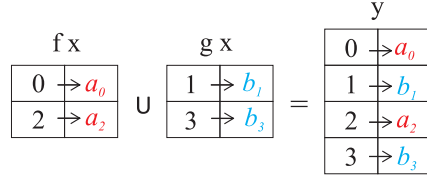


Figure 3.12: **MApply2Union** in *MHCOL*

With this change of data representation, we move away from the algebraic nature of Σ -*HCOL* towards a lower-level representation. In this representation, an actual value must be associated with a key in a dictionary before it can be accessed. Trying to access an uninitialized key is an error. It means that *MHCOL* operators could return errors and thus have the type: `mem_block \rightarrow option mem_block`. However, we will prove later that our translation of a structurally correct Σ -*HCOL* program produces an *MHCOL* program that does not err when applied to an input memory block that matches the expected input sparsity patterns.

Like in Σ -*HCOL*, we use *mixed embedding* [45] (a combination of *shallow* and *deep embedding*) to represent *MHCOL* operators. The following record type is used:

```

Record MSHOperator {i o: ℕ} : Type := mkMSHOperator {

    mem_op: mem_block → option mem_block;

    mem_op_proper: Proper ((equiv) ==> (equiv)) mem_op;

    m_in_index_set: FinNatSet i;

    m_out_index_set: FinNatSet o;

}.

```

Listing 3.8: MSHOperator definition

It is indexed by dimensions of input and output memory blocks. The fields include: a function implementing the operation on memory blocks which can fail (returning `None`); a *proper morphism* [46] instance for this function with respect to the setoid equality `equiv` (required because the carrier type is still abstract); and the two sets which define input and output memory access patterns.

Additionally, all *MHCOL* operator implementations must satisfy certain *memory safety* properties. We have formulated these properties as the typeclass, `MSHOperator_Facts`, and have proven instances of it for all operators. This is a similar approach to what we took with Σ -*HCOL structural properties*, but the properties are different:

1. When applied to a memory block which has mappings present for all keys in `m_in_index_set`, `mem_op` will not return an error.
2. The `mem_op` must assign a value to each element with a key in `m_out_index_set` and must not assign a value to any element with a key not in `m_out_index_set`.
3. The output block of `mem_op` is guaranteed to contain no mappings for keys outside of an operator’s declared output size.

3.3.1 Memory Model

MHCOL definitions rely on *memory block* abstraction which can be viewed as a part of a more comprehensive *memory model* used in later stages. In *MHCOL*, memory blocks are immutable and transient and are passed as arguments to the operators and returned as the results. However, the same type of memory block can be made persistent and organized into a *memory* which will maintain the state of a collection of such blocks, whereas each block will maintain the state of its cells. This hierarchical two-level memory organization was inspired by the CompCert [24] compiler [47] and by the Vellvm project.

As we re-use this memory model for other HELIX languages, it is generalized to support various value types. This is implemented using a Coq module system. All modules used in the memory model definition, along with their dependencies, are shown in Figure 3.13. The square boxes represent module types, while the rounded ones are module instances. Dashed lines depict type dependencies and solid lines signify subtyping.

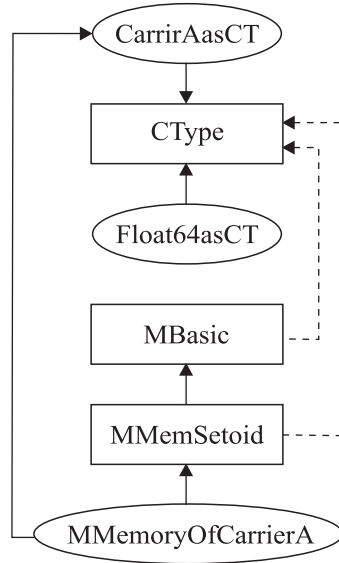


Figure 3.13: Memory model modules

The type of values was abstracted as a `CType` module type which can be instantiated for \mathcal{R} as `CarrierAasCT`. The full definition of this module is discussed in Section 3.4.1. Suffice it to say that it contains a type and the equality relation:

```

Module Type CType.

  Parameter Inline t : Type.

  Declare Instance CTypeEquiv: Equiv t.

  Declare Instance CTypeSetoid: @Setoid t CTypeEquiv.

  ...

End CType.

```

Listing 3.9: Partial `CType` module type definition

Parameterized by the module type `CType`, the module type `MBasic` defines memory model basics. It includes the abstract type `memory`, which represents a mapping from “addresses” (represented as natural numbers) to memory blocks and provides the essential operations for adding, removing, or looking up memory blocks.

It also provides a constant `memory_empty`, which represents the initial empty memory state. An abridged definition of `memory` interface from the `MBasic` module type is shown in Listing 3.10.

```

Definition memory : Type.

Definition mem_block : Type.

Definition memory_empty : memory.

Definition memory_lookup : memory → ℕ → option mem_block.

Definition memory_set : memory → ℕ → mem_block → memory.

Definition memory_remove : memory → ℕ → memory.

Definition memory_keys_lst : memory → list ℕ.

Definition memory_next_key : memory → ℕ.

Definition mem_block_exists : ℕ → memory → Prop.

Parameter decidable_mem_block_exists :
  ∀ (k : ℕ) (m : memory), decidable (mem_block_exists k m).

```

Listing 3.10: Memory interface

The memory address space is unbounded. In a given state, some addresses could be already associated with memory blocks while others may be not initialized yet. Operation `memory_set` assigns a memory block to a given memory address. It could be also used to change a memory state by replacing a block at the given address. Decidable predicate `mem_block_exists` checks whether a given memory address has been initialized. A block can be freed with `memory_remove`. The function `memory_next_key` returns the address of the next unallocated address.

The memory block has abstract type `mem_block`, which in turn is also a mapping of “offsets” (represented as natural numbers) to values of type `CType.t`.

Memory blocks are also unbounded and have interface similar to `memory`. A constant `mem_empty` represents an empty memory block which contains no values. An abridged definition of `mem_block` interface from the `MBasic` module is shown in Listing 3.11.

```

Definition mem_empty : mem_block.

Definition mem_lookup :  $\mathbb{N} \rightarrow \text{mem\_block} \rightarrow \text{option } \text{CType.t}$ .

Definition mem_add :  $\mathbb{N} \rightarrow \text{CType.t} \rightarrow \text{mem\_block} \rightarrow \text{mem\_block}$ .

Definition mem_delete :  $\mathbb{N} \rightarrow \text{mem\_block} \rightarrow \text{mem\_block}$ .

Definition mem_in :  $\mathbb{N} \rightarrow \text{mem\_block} \rightarrow \text{Prop}$ .

Parameter decidable_mem_in :  $\forall (k : \mathbb{N}) (m : \text{mem\_block}), \text{decidable } (\text{mem\_in } k m)$ .

Definition mem_keys_lst : mem_block  $\rightarrow \text{list } \mathbb{N}$ .

Definition mem_value_lst : mem_block  $\rightarrow \text{list } \text{CType.t}$ .

```

Listing 3.11: Memory block interface

The `MMemSetoid` module type extends `MBasic` with a definition of setoid equality for memory blocks and proofs of proper morphism for memory operations with respect to setoid equality. Finally, `MMemoryOfCarrierA` instantiates this module with `CarrierAasCT` for `CType`. An *MHCOL* language definition and all related proofs are done for this instantiation of a memory model.

3.3.2 Example

A result of a translation of the Σ -*HCOL* formulation of a dynamic window monitor from Listing 4.7 to *MHCOL* is shown in Listing 3.12. It has been abridged to hide the non-essential operator parameters.

```

Definition dynwin_MHCOL (a: avector 3) : @MSHOperator (1 + 4) 1 :=

```

```

MSHCompose (MSHBinOp (IgnoreIndex2 Zless))
(MHTSUMUnion plus
  (MSHCompose (MSHEmbed (le_S (le_n 1)))
    (MSHReduction 0 plus
      (λ jf : FinNat 3,
        MSHCompose
          (MSHCompose (MSHPointwise (Fin1SwapIndex jf (mult_by_nth a)))
            (MSHInductor (' jf) mult 1))
            (MSHPick (GathH1_domain_bound_to_base_bound (h_bound_first_half 1 4))))))
    (MSHCompose (MSHEmbed (le_n 2))
      (MSHReduction 0 minmax.max
        (λ jf : FinNat 2,
          MSHCompose
            (MSHBinOp
              (λ (i : FinNat 1) (a0 b : CarrierA),
                IgnoreIndex abs i (Fin1SwapIndex2 jf (IgnoreIndex2 sub) i a0 b)))
            (MSHIUnion
              (λ jf0 : FinNat 2,
                MSHCompose (MSHEmbed (proj2_sig jf0))
                  (MSHPick
                    (h_index_map_compose_range_bound
                      (GathH_jn_domain_bound (' jf) 2 (proj2_sig jf))
                      (h_bound_second_half 1 4) (' jf0)
                      (proj2_sig jf0))))))))))

```

Listing 3.12: Dynamic Window Monitor in *MHCOL*

3.4 *DHCOL* Language

DHCOL is an imperative language, deep-embedded in Coq proof assistant. The main data it operates on are fixed-size vectors of \mathcal{R} values, stored in memory. In addition to memory, *DHCOL* has lexically scoped variables, which could hold \mathcal{R} values, pointers to memory blocks, and natural numbers (used for loop bounds and memory offset computations).

All variables are immutable. The language is statically scoped and de Bruijn indices are used to reference variables from an *evaluation context* which holds the

values of all variables in scope, with the most recently introduced one at the top.

It should be noted that *DHCOL* is not a general purpose programming language. It is an intermediate representation language for the class of problems HELIX is designed for. As such, it is heavily focused on operations on \mathcal{R} vectors. Only the features needed to represent corresponding HELIX abstractions are present in the language. It may seem fairly esoteric, compared to general purpose programming languages.

The *DHCOL* memory model, shared with *MHCOL*, is described in Section 3.3.1. New memory blocks could be allocated and freed by a *DHCOL* program, and their elements can be modified repeatedly. This changing memory state represents an imperative aspect of the *DHCOL* design.

DHCOL supports typed *expressions*: *NExpr* for natural numbers, *AExpr* for \mathcal{R} values, *MExpr* for constant memory blocks, and *PExpr* for memory block pointers. \mathcal{R} and natural number expressions allow constants and provide a set of arithmetic operations like addition, subtraction, and division. Additionally, expressions could reference variables from the environment, which are typed. Evaluation of an expression has no side-effects but could fail.

3.4.1 *DHCOL* Type Parametrization

Like previous language in the HELIX transformation chain, *DHCOL* also uses abstract data type \mathcal{R} for data stored in memory and \mathbb{N} for memory addresses and offsets within memory blocks. However, this is the last language in the chain to use these types. The very next language in the chain, *FHCOL*, will be identical to *DHCOL* modulo these two types. *FHCOL* will use floating point numbers instead of \mathcal{R} and unsigned fixed-length machine integers instead of \mathbb{N} . Thus, *FHCOL* and

DHCOL are essentially a family of languages parameterized by different types. To implement such parametrization, we used Coq’s module system. We defined two module types, as shown in Listings 3.13 and 3.14, to wrap each type with required operations and properties.

```
Module Type CType.

  Parameter Inline t : Type.

  (* Equality *)
  Declare Instance CTypeEquiv: Equiv t.
  Declare Instance CTypeSetoid: @Setoid t CTypeEquiv.

  (* Values *)
  Parameter CTypeZero: t.
  Parameter CTypeOne: t.

  (* operations *)
  Parameter CTypePlus : t → t → t.
  Parameter CTypeNeg  : t → t.
  Parameter CTypeMult : t → t → t.
  Parameter CTypeAbs  : t → t.
  Parameter CTypeZLess: t → t → t.
  Parameter CTypeMin  : t → t → t.
  Parameter CTypeMax  : t → t → t.
  Parameter CTypeSub  : t → t → t.

  (* Proper *)
  Declare Instance Zless_proper: Proper ((=) ⇒ (=) ⇒ (=)) CTypeZLess.
  Declare Instance abs_proper: Proper ((=) ⇒ (=)) CTypeAbs.
  Declare Instance plus_proper: Proper((=) ⇒ (=) ⇒ (=)) CTypePlus.
```

```

Declare Instance sub_proper: Proper((=)  $\implies$  (=)  $\implies$  (=)) CTypeSub.
Declare Instance mult_proper: Proper((=)  $\implies$  (=)  $\implies$  (=)) CTypeMult.
Declare Instance min_proper: Proper((=)  $\implies$  (=)  $\implies$  (=)) CTypeMin.
Declare Instance max_proper: Proper((=)  $\implies$  (=)  $\implies$  (=)) CTypeMax.

End CType.

```

Listing 3.13: CType module type

For CType, we define equality and equivalence relations; constants for *additive identity* (zero) and *multiplicative identity* (one); and basic algebraic operations like addition and multiplication. Additionally, we require all these operations to be proper with respect to defined equality. CType is in essence a subset of what we had assumed and could easily be instantiated for \mathcal{R} . The set of properties is smaller since at this stage, we no longer require some algebraic properties like *ring* or *total order*. As we will see later in Section 3.5, this module could be also instantiated for IEEE floating point numbers.

```

Module Type NType.

  Parameter Inline t : Type.

  Declare Instance NTypeEquiv: Equiv t.
  Declare Instance NTypeSetoid: @Setoid t NTypeEquiv.

  (* Values *)
  Parameter NTypeZero: t.

  (* Decidable equality *)
  Declare Instance NTypeEqDec:  $\forall$  x y: t, Decision (x = y).

```

```

(* could always be converted to 'N' *)
Parameter to_N: t → ℕ.
Declare Instance to_N_proper: Proper ((=) ⇒ (=)) to_N.

(* not all ℕs could be converted to 't' *)
Parameter from_N: ℕ → err t.
Declare Instance from_N_proper: Proper ((=) ⇒ (=)) from_N.

(* arithmetics operators *)
Parameter NTypeDiv   : t → t → t.
Parameter NTypeMod   : t → t → t.
Parameter NTypePlus  : t → t → t.
Parameter NTypeMinus : t → t → t.
Parameter NTypeMult  : t → t → t.
Parameter NTypeMin    : t → t → t.
Parameter NTypeMax    : t → t → t.

Declare Instance NTypeDiv_proper: Proper ((=) ⇒ (=) ⇒ (=)) NTypeDiv.
Declare Instance NTypeMod_proper: Proper ((=) ⇒ (=) ⇒ (=)) NTypeMod.
Declare Instance NTypePlus_proper: Proper ((=) ⇒ (=) ⇒ (=)) NTypePlus.
Declare Instance NTypeMinus_proper: Proper ((=) ⇒ (=) ⇒ (=)) NTypeMinus.
Declare Instance NTypeMult_proper: Proper ((=) ⇒ (=) ⇒ (=)) NTypeMult.
Declare Instance NTypeMin_proper: Proper ((=) ⇒ (=) ⇒ (=)) NTypeMin.
Declare Instance NTypeMax_proper: Proper ((=) ⇒ (=) ⇒ (=)) NTypeMax.

Parameter to_string: t → String.string.

(* If [from_N] succeeds for a number, it also succeeds for all
   numbers less than it.
   *))
Parameter from_N_lt:

```



```

    ∀ x xi y,
      from_ℕ x ≡ inr xi →
      (y < x) % ℕ →
      ∃ yi, from_ℕ y ≡ inr yi.

    (* 0 is always convertible *)
    Parameter from_ℕ_zero: ∃ z, from_ℕ 0 ≡ inr z.

End NType.

```

Listing 3.14: NType module type

Our `NType` module type definition is similar to the `CType` with a few additions. A method to convert `NType.t` values to strings is provided mostly for debugging convenience. We additionally require this type to always be convertible to \mathbb{N} . Conversion from \mathbb{N} is also defined, but it could return an error, because for fixed-size parametrizations, it might not be possible to fit arbitrary-size natural numbers to a fixed number of bits. For error handling, we use `err` monad, described later. A couple of properties must be proven for instances of this module type. The first property states that if conversion from \mathbb{N} succeeds for a given natural number, it will also succeed for all natural numbers less than this. This property could be considered a form of monotonicity. The second property simply states that natural number 0 could be always successfully converted to `NType.t`. This makes the type inhabited, with at least one value corresponding to $0 \in \mathbb{N}$. This module type could easily be instantiated for natural numbers as well as for fixed-length machine integers.

It should be noted that while `NType.t` is used to represent loop indices in iterative operators, loop bounds are expressed as natural numbers. This implemen-

tation decision is meant to simplify proofs by induction.

A diagram showing the modular organization of *DHCOL* is shown in Figure 3.14. The square boxes represent module types, while the rounded ones are module instances. Dashed lines depict type dependencies and solid lines signify subtyping.

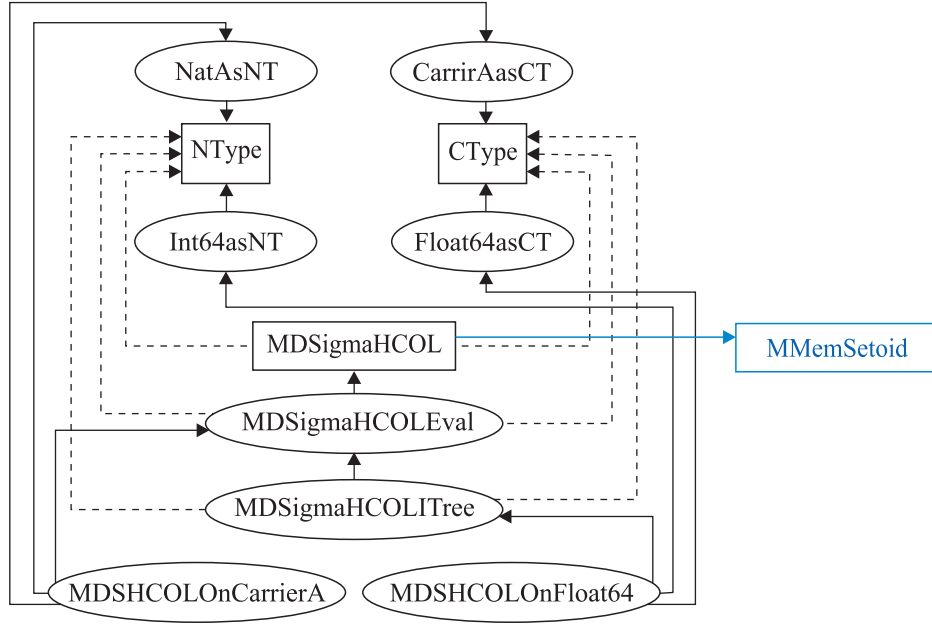


Figure 3.14: *DHCOL* modules

This diagram includes some modules which will be discussed later, in particular `MDSigmaHCOLTree` discussed in Section 4.7 and `Float64asCT`, `Int64asCT`, and `MDSHCOLOnFloat64` discussed in Section 3.5. The blue `MMemSetoid` block connects this diagram to the memory model shown in Figure 3.13. It reflects the fact that the *DHCOL* module type depends on a memory model module type, instantiated with the same `CType` and `NType` parameters.

3.4.2 *DHCOL* Expressions and Operators

The *DHCOL* language family is defined as a module type parametrized by modules **CT** for **CType** and **NT** for **NType**. It defines *DHCOL* expressions and operators. There are four expression types, each of which can be evaluated to the values of the corresponding types:

NExpr is a type of integer expressions which evaluates to **NType.t** values.

PExpr type represents pointers to blocks in memory. These pointers can be used to modify the objects they point to, changing the memory state. It evaluates to a tuple with type $\mathbb{N} \times \text{NT.t}$. The first element of the tuple is an address of a block in memory and the second is the size of this block.

MExpr also refers to memory blocks but can only be used to access, not modify data. In addition to referencing blocks in memory, it can also represent standalone constant memory blocks. It evaluates to a tuple with type $\text{mem_block} \times \text{NT.t}$.

AExpr expressions evaluate to scalar values of **CType.t**.

The definition of **AExpr** type is shown in Listing 3.15. The evaluation semantics for expressions will be discussed later in Section 3.4.3.

```
Inductive AExpr : Type :=
| AVar   : var_id → AExpr
| AConst : CT.t → AExpr
| ANth   : MExpr → NExpr → AExpr
| AAbs   : AExpr → AExpr
| APlus  : AExpr → AExpr → AExpr
| AMinus : AExpr → AExpr → AExpr
| AMult  : AExpr → AExpr → AExpr
| AMin   : AExpr → AExpr → AExpr
```

```

| AMax  : AExpr → AExpr → AExpr
| AZless: AExpr → AExpr → AExpr.

```

Listing 3.15: AExpr type

The *DHCOL* operator type is shown in Listing 3.16. In addition to the expression types described above, it uses the *MemRef* type, which is a memory pointer combined with an offset as a natural number: $(PE\text{Expr} \times N\text{Expr})$.

```

Inductive DSHOperator :=
| DSHNop
| DSHAssign (src dst: MemRef)
| DSHIMap (n: ℕ) (x_p y_p: PExpr) (f: AExpr)
| DSHBinOp (n: ℕ) (x_p y_p: PExpr) (f: AExpr)
| DSHMemMap2 (n: ℕ) (x0_p x1_p y_p: PExpr) (f: AExpr)
| DSHPower (n:NExpr) (src dst: MemRef) (f: AExpr) (initial: CT.t)
| DSHLoop (n:N) (body: DSHOperator)
| DSHAlloc (size:NT.t) (body: DSHOperator)
| DSHMemInit (y_p: PExpr) (value: CT.t)
| DSHSeq (f g: DSHOperator).

```

Listing 3.16: DSHOperator type

A detailed description of all DHCOL operators follows.

DSHNop

This is a “no-op” operator. It does nothing.

DSHAssign (src dst: MemRef)

An assignment operator copies a single `CType.t` value from memory location *src* to memory location *dst*.

DSHIMap (n: N) (x y: PExpr) (f: AExpr)

It iterates the index from 0 to $n - 1$. For each iteration, the values from **x** at the index offset are added to the evaluation context, and the expression **f** is evaluated. It could be viewed as calling function f with the two arguments: an index i of type `NType.t` and a value **x**[i] of type `CType.t`. The result is written to **y**[i].

DSHBinOp (n: N) (x y: PExpr) (f: AExpr)

This operator is similar to *DSHIMap*. It also iterates from 0 to $n - 1$. For each iteration, it reads two values: at the index offset and at n plus the index offset from **x** and adds them to the evaluation context and then evaluates the expression **f**. It could be viewed as calling the function f with the two arguments: **x**[i] and **x**[$n+i$]. The result is written to **y**[i].

DSHMemMap2 (n: N) (x0 x1 y: PExpr) (f: AExpr)

This is another iterative map. It also iterates from 0 to $n - 1$. For each iteration, it reads **x0**[i] and **x1**[i] and adds them to the evaluation context and evaluates the expression **f**. It could be viewed as calling the function f with the two arguments: **x0**[i] and **x1**[i]. The result is written to **y**[i].

DSHPower (n: NExpr) (src dst: MemRef) (f: AExpr) (initial: CT.t)

First, it initializes a memory location pointed by **dst** with **initial** value and proceeds to iterate n times. On each iteration, values are loaded from **src** and **dst**, the **f** is evaluated, and the result of the evaluation is stored back in **dst**. In C-like pseudo code, it could be expressed as: `*dst=initial; while(n--){*dst=f(*src,*dst)}`.

DSHLoop (n:ℕ) (body: DSHOperator)

This is a simple loop. It evaluates a body n times. For each iteration, the value of the loop index (starting from 0) is put on the top of the evaluation context before evaluating the *body*.

DSHAlloc (size:NT.t) (body: DSHOperator)

Lexically scoped memory allocation. It allocates a new memory block of size **size**. The newly allocated block is uninitialized. The new block's offset in memory is put on the top of the evaluation context and then the **body** is evaluated. The block is only available while the body is being evaluated.

DSHMemInit (y: PExpr) (value: CT.t)

Initialize all elements of block **y** with given **value**.

DSHSeq (f g: DSHOperator)

Evaluate f and then, evaluate g . Memory modifications performed by the evaluation of the first operator are visible during the evaluation of the second.

3.4.3 *DHCOL* Evaluation

Recursive definition *evalDSHOperator* takes an operator to evaluate, an evaluation context σ , and the initial memory state **mem**. If successful, it returns the final memory state after the evaluation:

```
Fixpoint evalDSHOperator (σ: evalContext) (op: DSHOperator)
  (mem: memory) (fuel: ℕ): option (err memory).
```

It is implemented via structural recursion over the structure of the *DHCOL* expression. For error handling, it is wrapped in *exception monad*, for which we use **err** type defined by Vellvm. All *DHCOL* programs are guaranteed to terminate, but *fuel* is used to simplify termination proofs. The evaluation result is double wrapped in an *option monad* (on top of the **err** monad) to account for errors due to insufficient fuel. There is a matching **estimateFuel** function which estimates the fuel required to execute a given *DHCOL* operator, and we’ve proven a lemma showing that the estimated fuel is always sufficient for a successful **evalDSHOperator** completion. In other words, with estimated fuel it will never return **None** (but may still return an error).

The *evalDSHOperator* function defines big-step operation semantics, formally presented in Appendix A.

3.4.4 Example

The result of a translation of the *MHCOL* formulation of a dynamic window monitor from Listing 3.12 to *DHCOL* is shown in Listing 3.17.

```

Definition dynwin_DHCOL : DSHOperator :=
DSHAlloc 2
  (DSHSeq
    (DSHAlloc 2
      (DSHAlloc 2
        (DSHSeq
          (DSHSeq
            (DSHAlloc 1
              (DSHSeq
                (DSHAlloc 1
                  (DSHSeq (DSHMemInit 1 (PVar 0) CarrierAz)
                    (DSHLoop 3
                      (DSHSeq
                        (DSHAlloc 1
                          (DSHSeq
                            (DSHAssign

```

```

(PVar 9, NConst 0)
(PVar 0, NConst 0))
(DSHalloc 1
  (DSHSeq
    (DSHPower
      (NVar 2)
      (PVar 1, NConst 0)
      (PVar 0, NConst 0)
      (AMult (AVar 1) (AVar 0))
      CarrierA1)
    (DSHIMap 1
      (PVar 0)
      (PVar 4)
      (AMult
        (AVar 0)
        (Anth (MVar 10) (NVar 4))))))
    (DSHMemMap2 1 (PVar 1)
      (PVar 2) (PVar 2)
      (APlus (AVar 1) (AVar 0))))))
  (DSHAssign (PVar 0, NConst 0) (PVar 1, NConst 0)))
(DSHalloc 1
  (DSHSeq
    (DSHalloc 1
      (DSHSeq (DSHMemInit 1 (PVar 0) CarrierAz)
        (DSHLoop 2
          (DSHSeq
            (DSHalloc 2
              (DSHSeq
                (DSHLoop 2
                  (DSHalloc 1
                    (DSHSeq
                      (DSHAssign
                        (
                          PVar 11,
                          NPlus
                          (NPlus
                            (NConst 1)
                            (NMult (NVar 3) (NConst 1)))
                            (NMult
                              (NVar 1)
                              (NMult (NConst 2) (NConst 1))))
                          (PVar 0, NConst 0))
                      (DSHAssign

```



```

(PVar 0, NConst 0)
(PVar 2, NVar 1))))
(DSHBinOp 1
  (PVar 0)
  (PVar 3)
  (AAbs (AMinus (AVar 1) (AVar 0))))))
(DSHMemMap2 1 (PVar 1)
  (PVar 2) (PVar 2)
  (AMax (AVar 1) (AVar 0))))))
(DSHAssign (PVar 0, NConst 0) (PVar 2, NConst 1))))
(DSHMemMap2 2 (PVar 0) (PVar 1) (PVar 2)
  (APlus (AVar 1) (AVar 0))))))
(DSHBinOp 1 (PVar 0) (PVar 2) (AZless (AVar 1) (AVar 0)))

```

Listing 3.17: Dynamic Window Monitor in *DHCOL*

3.5 *FHCOL* Language

FHCOL language is an instantiation of *DHCOL*, parameterized with machine numeric types. This brings us one abstraction step down from *DHCOL* toward machine code.

We instantiate module type `NType` to represent unsigned 64-bit machine integers as an `MInt64asNT` module. We use the integer type definition from the Vellvm library, which can be traced back to a similar CompCert definition. In this implementation, a machine integer (type `int`) is represented as a Coq arbitrary-precision integer (type `Z`) plus the proof that it is in the range from 0 (included) to `modulus` (excluded):

```
Record int: Type := mkint { intval: Z; intrange: -1 < intval < modulus }.
```

For example, the concrete type `Int64` that we use, is an instantiation of `int` with `modulus = 264`. Definitions of the remaining fields of the `NType` module and required proofs are straightforward. To extend HELIX to generate code for other

hardware platforms, `int` could be similarly instantiated for 32-bit or even 16-bit integers.

We instantiate module type `CType` to represent fixed-length 64-bit IEEE floating point numbers as an `MFloat64asCT` module. To work with IEEE floating point numbers, we use the Flocq [48] library, which provides comprehensive formalization of IEEE floating point arithmetic in Coq. Flocq supports a range of binary float representations, parameterized by two constants: `P` and e_{max} describing the range of the integer significands and exponents, respectively. `P` is the number of bits of the mantissa including the implicit one. The e_{max} is chosen so that $2^{e_{max}}$ is the smallest power of two that is too large to be represented. Finite non-zero numbers are encoded in the form $m \times 2^e$ where m and e represent the respective integer values of the mantissa (significand) and the exponent, each bounded by `P` and e_{max} , respectively. To encode *exceptional values* (such as infinities) the following inductive type definition is used in Flocq:

```
Inductive binary_float :=
  | B754_zero (s :  $\mathbb{B}$ )
  | B754_infinity (s :  $\mathbb{B}$ )
  | B754_nan (s :  $\mathbb{B}$ ) (pl : positive) : nan_pl pl = true → binary_float
  | B754_finite (s :  $\mathbb{B}$ ) (m : positive) (e :  $\mathbb{Z}$ ) : bounded m e = true → binary_float.
```

Flocq’s `binary64` type which corresponds to the `double` type in C language is defined with `P = 24` and $e_{max} = 128$. We instantiate the `CType` module type for the `binary64` type, which represents 64-bit IEEE floating point numbers. As with integers, definitions of all required module fields are straightforward. Whenever the rounding mode needs to be specified, we always use “round to nearest.”

One operation whose definition is non-obvious is `CTypeZLess`. It must compare two floating point numbers and return `one` if the first is less than the second

or `zero` otherwise. Flocq `Bcompare` implements the comparison function, as defined in the IEEE specification. It is fairly complex and contains 32 lines of match statements for various combinations of IEEE binary value types: *zero*, *infinities*, *finite non-zero numbers*, and `NaNs`. One point of interest here is the `NaN` handling. In this implementation, it differs from how `NaNs` are handled by the `fcmp olt` instruction in LLVM IR. From a HELIX system point of view, this is inconsequential, the proofs are constrained to input data which contains no `NaNs`, and we can prove that `NaNs` will not appear as the result of any computations. Thus, to simplify *FHCOL* to LLVM IR compiler proofs, we handle `NaN` values during the comparison as similar as possible to how it is done in the IR `fcmp olt` instruction, into which `CTypeZLess` will be compiled. Per IR `fcmp olt` specification, the expected behavior is to “*yield true if both operands are not a QNaN and the first operand is less than the second operand,*” and this is how we define `CTypeZLess` on `binary64`:

```

Definition CTypeZLess (a b: binary64) : binary64 :=
  match a, b with
  | B754_nan _ _ _ _ , _ | _, B754_nan _ _ _ _ => CTypeZero
  | _, _ =>
    match Bcompare _ _ a b with
    | Some Datatypes.Lt => CTypeOne
    | _ => CTypeZero
    end
  end.

```

One may notice that our definition differs from the literal interpretation of IR spec. In our definition, we make no distinction between “quiet” and “signalling” `NaNs`. In IR specification, at of time of this writing, the behavior for the `SNaN` under comparison is not specified. We believe this is just a documentation shortcoming. To verify this, we tested how `fcmp` instruction behaves in an LLVM IR compiler

reference implementation for an x86 backend and indeed, it does not distinguish between QNaN and SNaN. This is the behavior we assume in HELIX.

The final step to implementing *FHCOL* is a matter of simple module instantiation using MFloat64asCT and MInt64asNT modules:

```
Module Export MDSHCOLONFloat64 := MDSigmaHCOLIITree(MFloat64asCT)(MInt64asNT).
```

3.5.1 Example

A result of the translation of a *DHCOL* formulation of the dynamic window monitor from Listing 3.17 to *FHCOL* is shown in Listing 3.18.

```
Definition dynwin_FHCOL : DSHOperator :=
DSHAlloc 2%int64
  (DSHSeq
    (DSHSeq
      (DSHAlloc 1%int64
        (DSHSeq
          (DSHSeq (DSHMemInit (PVar 0) Float64asCT.Float64Zero)
            (DSHAlloc 1%int64
              (DSHLoop 3
                (DSHSeq
                  (DSHAlloc 1%int64
                    (DSHSeq
                      (DSHAssign
                        (PVar 7, NConst 0)
                        (PVar 0, NConst 0))
                      (DSHAlloc 1%int64
                        (DSHSeq
                          (DSHPower (NVar 2)
                            (PVar 1, NConst 0)
                            (PVar 0, NConst 0)
                            (AMult (AVar 1) (AVar 0))
                            Float64asCT.Float64One)
                          (DSHIMap 1 (PVar 0) (PVar 3)
                            (AMult (AVar 0)
                              (ANth
                                (MPtrDeref (PVar 8))
                                (NVar 4))))))))))
```

```

        (DSHMemMap2 1 (PVar 2) (PVar 1) (PVar 2)
          (APlus (AVar 1) (AVar 0))))))
      (DSHAssign (PVar 0, NConst 0)
        (PVar 1, NConst 0)))
    (DSHAlloc 1%int64
      (DSHSeq
        (DSHSeq (DSHMemInit (PVar 0) Float64asCT.Float64Zero)
          (DSHAlloc 1%int64
            (DSHLoop 2
              (DSHSeq
                (DSHAlloc 2%int64
                  (DSHSeq
                    (DSHLoop 2
                      (DSHAlloc 1%int64
                        (DSHSeq
                          (DSHAssign
                            (PVar 9,
                              NPlus
                                (NPlus
                                  (NConst 1)
                                  (NMult (NVar 3)
                                    (NConst 1)))
                                (NMult (NVar 1)
                                  (NMult
                                    (NConst 2)
                                    (NConst 1))))
                            (PVar 0,
                              NConst 0))
                          (DSHAssign
                            (PVar 0,
                              NConst 0)
                            (PVar 2, NVar 1))))
                        (DSHBinOp 1 (PVar 0) (PVar 2)
                          (AAbs (AMinus (AVar 1) (AVar 0))))))
                      (DSHMemMap2 1 (PVar 2) (PVar 1) (PVar 2)
                        (AMax (AVar 1) (AVar 0))))))
                    (DSHAssign (PVar 0, NConst 0)
                      (PVar 1, NConst 1))))
                (DSHBinOp 1 (PVar 0) (PVar 2) (AZless (AVar 1) (AVar 0)))

```

Listing 3.18: Dynamic Window Monitor in *FHCOL*

Chapter 4

HELIX Verification

In this section, we describe how the HELIX translation steps are implemented and verified. Due to the disparate nature of the languages involved, we use a variety of techniques to implement and prove the different steps.

The sequence of verification steps in HELIX is shown in Figure 4.1 and briefly described below and again in more detail in subsequent sections.

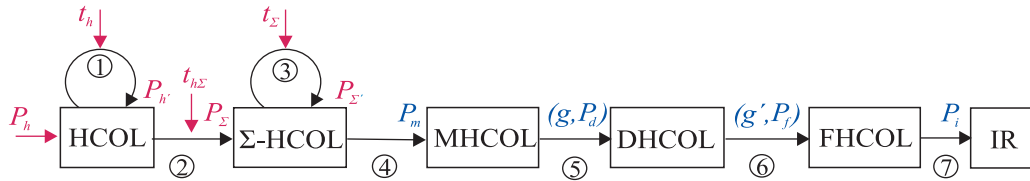


Figure 4.1: HELIX chain of verification

Artefacts provided by SPIRAL are shown in red and HELIX-generated artefacts are shown in blue:

P_h - source program in *HCOL*

t_h - SPIRAL trace containing list of “breakdown” steps

$P_{h'}$ - program in *HCOL* after “breakdown” step (generated by SPIRAL)

$t_{h\Sigma}$ - SPIRAL trace containing list of OL to Σ - OL translation steps

P_Σ - program in Σ - $HCOL$ (generated by SPIRAL)

t_Σ - SPIRAL trace containing list of Σ - OL rewriting steps

$P_{\Sigma'}$ - program in Σ - $HCOL$ after rewriting step (generated by SPIRAL)

P_m - program in $MHCOL$

P_d - program in $DHCOL$

g - list of global variables upon which P_d depends along with their $DHCOL$ types

P_f - program in $FHCOL$

g' - list of global variables upon which P_f depends along with their $FHCOL$ types

P_i - program in LLVM IR

The numbered arrows in Figure 4.1 depict the translation steps, listed below along with brief descriptions of how they are validated:

- ① The $HCOL$ “breakdown” step constitutes application of a sequence of semantics-preserving rewriting steps from t_h to P_h resulting in $P_{h'}$. Each element of t_h corresponds to a single SPIRAL OL breakdown rule application. For each such rule, we formulate and prove a lemma. The semantic equivalence of P_h and $P_{h'}$ is then proven by automated sequential application of breakdown rule lemmas. See Section 4.1 for details.
- ② The $HCOL$ to Σ - $HCOL$ translation step constitutes application of a sequence of semantics-preserving rewriting steps from $t_{h\Sigma}$ to $P_{h'}$ lifted to Σ - $HCOL$ resulting in a full native (without lifted operators) Σ - $HCOL$ version P_Σ . Each element of $t_{h\Sigma}$ corresponds to a single SPIRAL Σ - OL rewriting rule application. For each such rule, we formulate and prove a lemma. The semantic equivalence

lence of lifted $P_{h'}$ and P_Σ is then proven by automated sequential application of rewriting rule lemmas. Additionally, we prove the *structural correctness* of the resulting expression. See Section 4.2 for details.

- ③ After the initial *HCOL* to Σ -*HCOL* translation, an additional translation step is performed by applying another series of rewriting rules from $t_{h\Sigma}$ to P_Σ resulting in $P_{\Sigma'}$. It is proven in the exact same manner as the previous step. See Section 4.3 for details.
- ④ The *MHCOL* program P_m is generated from $P_{\Sigma'}$ with the help of a Template-Coq metaprogram. For the resulting program, an `MSHOperator_Facts` instance is proven using proof automation. Then, the semantics preservation property is validated by proving an instance of an `SH_MSH_Operator_compat` typeclass for a P_m and $P_{\Sigma'}$ pair. This automated proof relies on `MSHOperator_Facts` as well as a proof of the structural correctness of the $P_{\Sigma'}$ generated during the previous step. See Section 4.4 for details.
- ⑤ The *DHCOL* program P_d is generated from P_m with the help of a Template-Coq metaprogram, which also produces a list g of the global variables P_d depends upon. Then, the `DSH_pure` and `MSH_DSH_compat` typeclass instances are proven using proof automation. See Section 4.5 for details.
- ⑥ The *DHCOL* to *FHCOL* translation is implemented in Gallina. It translates program P_d and a list of global variables g to the corresponding P_f and g' . This step is not validated. See Section 4.6 for discussion of our reasoning.
- ⑦ The final step of translation from *FHCOL* to LLVM IR is performed using a certified compiler that we wrote in Gallina. We have proven this compiler to be correct (with caveats). See Section 4.7 for details.

Thus, given an original *HCOL* expression, a SPIRAL trace file containing the transformation steps, and the intermediate SPIRAL code synthesis results (all shown in red in Figure 4.1), HELIX will either fail or provide the following high-level results:

1. LLVM IR version of the *HCOL* program, which includes SPIRAL-guided code optimizations.
2. Correctness guarantee that the IR program on IEEE floating-point inputs that do not contain NaN values will generate the same results as the *HCOL* program on real numbers up to guarantees provided by the numeric analysis step.

If HELIX fails, it could be that SPIRAL-generated steps and intermediate results are either inconsistent, use unproved rules, or that the HELIX LLVM compiler is missing the functionality required to compile the given expression. The first indicates a bug in the SPIRAL system, which needs to be fixed. The latter two reasons point to HELIX implementation shortcomings that could be cured by extending HELIX with additional rules or the compiler capabilities using the existing framework.

We discuss how these results are achieved in subsequent sections.

4.1 *HCOL* Breakdown

At the first translation step, HELIX performs semantically preserving modifications of the *HCOL* expressions. The goal is to break down more complex operations into elementary ones, representing a fully terminated computation dataflow graph optimized for target hardware taking into account such target architecture parameters

as number of words in SIMD instructions, number of cores, and CPU cache size. The breakdown steps are determined by SPIRAL and validated by HELIX. Each step is implemented in SPIRAL as an application of a “breakdown rule.” Per the translation validation approach discussed earlier, we prove a lemma for each rule and apply them following the trace generated by SPIRAL. Application of the breakdown rules is done using *setoid rewriting* [49] together with *HCOL* operator equational theory.

The result of the *HCOL* rewriting steps of the expression from Listing 3.2 is shown in Listing 4.1

```

Definition dynwin_HCOL (a: avector 3) : : avector (1 + (2 + 2)) → avector 1 :=
  HBinOp (IgnoreIndex2 Zless) ∘
    HCross
      (HReduction plus zero ∘
        (HBinOp (IgnoreIndex2 mult) ∘ HPrepend a) ∘
        HInduction _ mult one)
      (HReduction minmax.max zero ∘
        (HPointwise (IgnoreIndex abs)) ∘
        HBinOp (o:=2) (IgnoreIndex2 sub)).

```

Listing 4.1: Dynamic Window Monitor in *HCOL* after rewriting

It corresponds to the dataflow graph shown in Figure 4.2.

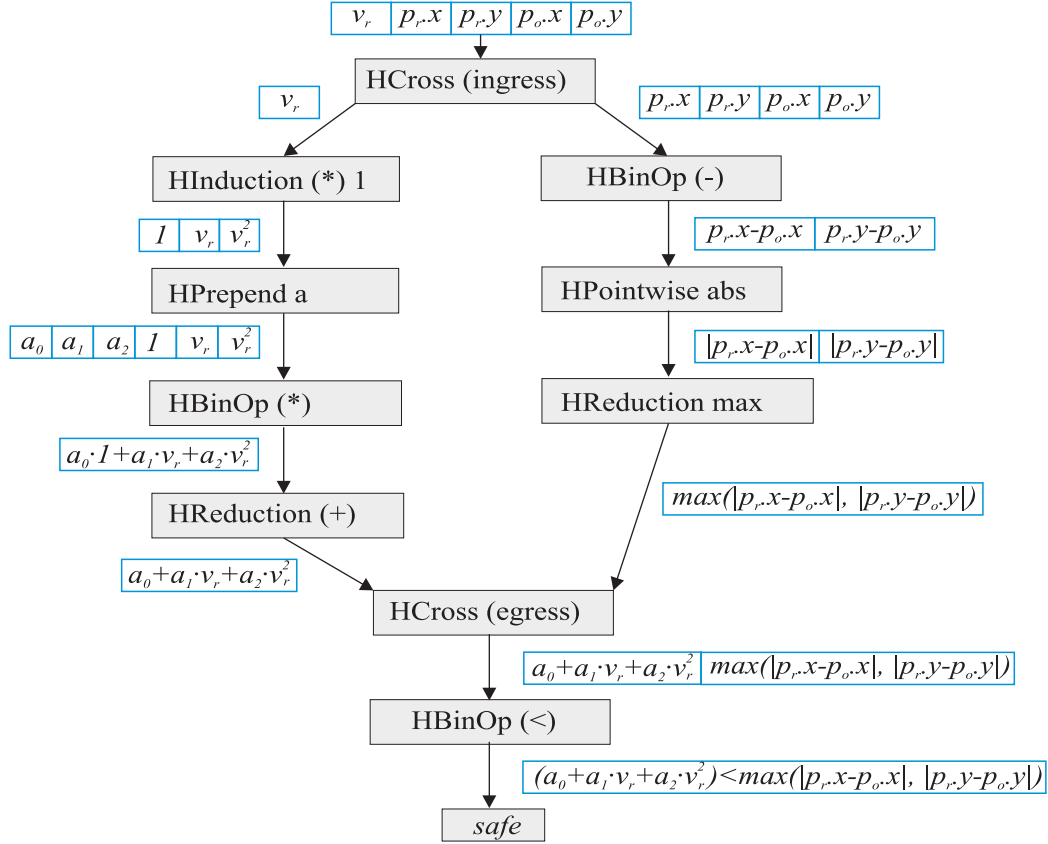


Figure 4.2: Dynamic Window Monitor dataflow graph in *HCOL*

4.1.1 Breakdown Rules as Lemmas

SPIRAL breakdown rules are expressed as lemmas in HELIX. Each lemma states the equality (using `equiv` relation) between two *HCOL* expressions. Typically, the lemma is used in a left-to-right direction in the `setoid_rewrite` Coq tactic, replacing the left hand side expression with the right hand side equivalent. Let us look at a couple of rules.

HScalarProd breakdown rule

Recall from Section 3.1, the `HScalarProd` operator calculates the dot product of two vectors. The input vectors are concatenated and passed as a single vector of size $n + n$. The result is returned as a single-element vector. For an input vector $\vec{x} = [x_0, x_1, \dots, x_{n+n-1}]$, it computes $[x_0 \cdot x_n + x_1 \cdot x_{n+1} + \dots + x_{n-1} \cdot x_{n+n-1}]$.

The SPIRAL breakdown rule for this operator states that it could be represented as a composition of `HReduction` and `HBinOp` operators. First, `HBinOp` is applied multiplying corresponding elements in the first and second halves of the input vector producing, as an intermediate result, a vector of size n with values $[x_0 \cdot x_n, x_1 \cdot x_{n+1}, \dots, x_{n-1} \cdot x_{n+n-1}]$. The operator takes as parameter f a function with type $\mathbb{I}_n \rightarrow \mathcal{R} \rightarrow \mathcal{R} \rightarrow \mathcal{R}$ which will be applied to an index and a pair of elements from the first and the second halves of the vector. Since in this case, we only want to multiply elements without using the index, we wrap the multiplication function in `IgnoreIndex2`, which discards the first argument, and use it as the parameter f .

Next, with `HReduction (+) 0`, we compute $[0 + x_0 \cdot x_n + x_1 \cdot x_{n+1} + \dots + x_{n-1} \cdot x_{n+n-1}]$ as a right fold. The corresponding lemma and the definition of the helper function are shown in Listing 4.2.

```

Definition IgnoreIndex2 {A B:Type} (f:A → A → A) := const (B:=B) f.

Fact breakdown_OScalarProd: ∀ {n:ℕ},
  HScalarProd (n:=n) = HReduction (+) 0 ∘ HBinOp (IgnoreIndex2 mult).

```

Listing 4.2: `HScalarProd` breakdown rule

HEvalPolynomial breakdown rule

The rule for `HEvalPolynomial` breakdown states that it could be broken down as a composition of `HScalarProd`, `HPrepend`, and `HMonomialEnumerator`.

Recall from Section 3.1, `HEvalPolynomial` computes a univariate polynomial n -th degree. It is parameterized by a vector of constant coefficients. The input and output scalar values are represented as vectors of size 1. For input $\vec{x} = [x_0]$ and parameter $\vec{a} = [a_0, a_1, \dots, a_n]$, it computes $[a_0 + a_1 \cdot x_0 + a_2 \cdot x_0^2 + \dots + a_n \cdot x_0^n]$.

In the broken down version, we first apply `HMonomialEnumerator` to the input which computes a vector of its powers: $[1, x_0, x_0^2, \dots, x_0^{n-1}]$. Then, we prepend this vector with elements of a using the `HPrepend` operator resulting in $[a_0, a_1, \dots, a_n, 1, x_0, x_0^2, \dots, x_0^{n-1}]$. Finally, we use `HScalarProd` to compute the dot product of the first and second halves of this vector resulting in $[a_0 \cdot 1 + a_1 \cdot x_0 + a_2 \cdot x_0^2 + \dots + a_n \cdot x_0^n]$. The corresponding lemma is shown in Listing 4.3.

Fact `breakdown_OEvalPolynomial`: $\forall (n:\mathbb{N}) (a: \text{avector } (S \ n)),$
 $\text{HEvalPolynomial } a = \text{HScalarProd} \circ \text{HPrepend } a \circ \text{HMonomialEnumerator } n.$

Listing 4.3: `HEvalPolynomial` breakdown rule

4.1.2 Semantics-preserving rewriting

We define our semantics preservation property as an equivalence relation on *HCOL* expressions. To prove that *HCOL* expression A could be broken down into *HCOL* expression B while preserving its semantics, we need to prove $A = B$.

In the case of simple operators, we can just prove a lemma stating the equality of the two exact expressions. For complex expressions consisting of a composition of multiple operators, such proof can be performed in a series of automated steps. Each step corresponds to an application of a “breakdown rule” modifying all or a

part of an expression. For each rule, there is a lemma in the form $A = B$. It is applied using the `setoid_rewrite` tactic, which searches the current expression for patterns matching A and replaces their occurrences with B . Because $(=)$ relation is transitive, proving each rewriting step will guarantee the equality between the initial expression and the results of an application of a sequence of rules. The rewriting rules in the HELIX library must be manually proven once but after that, these proofs can be reused to automatically prove the correctness of any sequence of their applications.

For example, to prove that $A \circ B = D \circ E$, we may first apply rule $A = D$ to get $D \circ B = D \circ E$ and then apply rule $B = E$ to $D \circ E = D \circ E$, which is true because our equality is reflexive.

To make this machinery work, we need to impose some additional requirements on operators. This is done by making them all instances of the `HOperator` typeclass:

```
Class HOperator {i o:N} (op: vector  $\mathcal{R}$  i  $\rightarrow$  vector  $\mathcal{R}$  o) :=
  op_proper :> Proper ((=)  $\implies$  (=)) op.
```

Listing 4.4: `HOperator` class

Currently, this typeclass does not contain any additional fields except the one it inherits from the `Proper` typeclass, which is required for the `setoid_rewrite` tactic to work. The theory of generalized setoid rewriting and related typeclasses is discussed in [49]. Informally, it could be said that this `Proper` typeclass instance guarantees that for any two inputs of an operator that are related (via `equiv` relation), the results of respective applications of the operator to these inputs will also be in the same relation.

For example, the listing below shows a proof of the `HOperator` typeclass

instance for the `HPointwise` operator from Listing 3.1. The proof involves applying a couple of helper utility lemmas (`Vforall2_intro_nth`, `Vbuild_nth`) and the existing `Proper` typeclass instance for `Vnth`.

```

Instance HPointwise_HOperator
  {n: ℕ}
  {f: ℤn → ℝ → ℝ}
  {pF: !Proper ((=) ⇒ (=) ⇒ (=)) f}:
  HOperator (@HPointwise n f).
Proof.
  intros x y E.
  apply Vforall2_intro_nth.
  intros j jc.
  unfold HPointwise.
  setoid_rewrite Vbuild_nth.
  apply pF.
  reflexivity.
  apply Vnth_proper, E.
Qed.

```

Listing 4.5: `HOperator` instance for `HPointwise`

Other breakdown rule proofs frequently make use of the algebraic properties of \mathcal{R} and linear algebra identities.

4.1.3 *HCOL* Semantics Preservation Verification Framework

The key techniques of our semantics preservation verification framework for *HCOL* rewriting are:

- Abstract the data type on which *HCOL* operates as carrier type \mathcal{R} .
- Assume an equivalence relation $(=)$ on \mathcal{R} .

- Assume some algebraic properties of \mathcal{R} .
- Define $(=)$ on vectors of \mathcal{R} as a pointwise relation.
- Define *HCOL* operators as functions from vectors to vectors (of a carrier type) which are instances of the `HOperator` typeclass.
- Define extensional equality of *HCOL* operators.
- Define breakdown rules as lemmas stating equality between *HCOL* expressions.

Using this framework, given the original and the final *HCOL* expressions, h and h' , and the trace (list) of breakdown rules applied to get from h to h' , the HELIX *HCOL* rewriting proof engine can prove that an applied sequence of breakdown rules is *semantically preserving* and that $h = h'$.

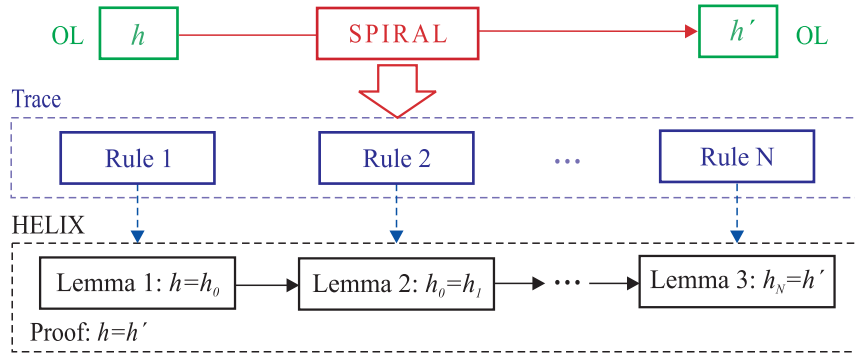


Figure 4.3: HELIX translation validation of SPIRAL *HCOL* rewriting

This approach, shown in Figure 4.3, is an extension of the *translation validation* introduced in Section 2.5. A sequence of rewriting steps is generated outside of HELIX by SPIRAL. Instead of proving that SPIRAL will always transform an expression correctly, HELIX formally verifies the correctness of the produced results. Given that SPIRAL and HELIX use the same library of breakdown rules, the proof of the goal $h = h'$ is a sequence of applications of setoid rewrites using already

proven per-rule lemmas from the HELIX library. We can automatically generate such proof from the trace and if Coq accepts it, the rewriting is proven correct. If, for some reason, the trace contains a non-semantically preserving rewriting sequence, Coq will not accept the proof.

4.2 *HCOL* to Σ -*HCOL* Translation

As mentioned in Section 3.2, *HCOL* operators can be embedded in Σ -*HCOL* using the `liftM_HOperator` wrapping operator. When lifting, we use the `Monoid_RthetaFlags` monoid to detect potential collisions. Thus, a trivial translation of an *HCOL* expression `h` to Σ -*HCOL* is simply `(liftM_HOperator Monoid_RthetaFlags h)`.

However, this first naïve translation can be further refined by the additional rewriting steps which replace *HCOL* operators with similar Σ -*HCOL* equivalents (e.g. lifted `HBinOp` with `SHBinOp`). To do that for complex *HCOL* operators, we need to exploit the facts that `liftM_HOperator` distributes over operator composition, and that the operator composition is associative. Translating a Σ -*HCOL* expression to a “normal form” (without occurrences `liftM_HOperator`) is the goal of the *HCOL* to Σ -*HCOL* translation step.

The reasoning about semantics preservation during Σ -*HCOL* rewriting is similar in approach to our reasoning about *HCOL* rewriting. The main difference is that Σ -*HCOL* operators work on sparse rather than dense vectors, and a different equality relation is used (see 3.2.7).

Finally, we need to make sure that the resulting Σ -*HCOL* expression is *structurally correct*, as discussed in Section 3.2.5. This is done by proving an instance of the `SHOperator_Facts` typeclass for the resulting expression. As mentioned earlier,

structural correctness proofs are compositional and thus easy to automate. Provided that we have instances of `SHOperator_Facts` for all basic operators, obtaining structural correctness for a composite operator is simply a matter of applying respective typeclass instances. We do this with simple *Ltac* automation. While automation solves all goals related to `SHOperator_Facts` instances, some of these goals introduce additional obligations which must also be proven. These have not been fully automated yet, but they could be in future, as discussed in Section 5.3.3.

In addition to semantics preservation and structural correctness, there are some additional properties which we want to verify for the final Σ -*HCOL* expression:

Sparsity contract “subtyping.” It guarantees that the resulting expression’s `in_index_set` is included in the original expression’s `in_index_set`, while the `out_index_set` of each expression is the same. This permits potential optimization (dead code elimination) during rewriting, when indices of input vectors which were used by the original expression are no longer used by the resulting expression. This is proven compositionally by constructing respective sparsity contracts of input and output expressions.

Totality of the computation. In general, Σ -*HCOL* operators work on sparse vectors. However, the sparsity is used only internally to represent partial computation. The whole composite computation should be total by taking the dense input and producing the dense output. That means that for top-level Σ -*HCOL* expressions, we want to prove that both `in_index_set` and `out_index_set` are the full sets. This is proven compositionally as well, by constructing respective sparsity contracts of the input and output expressions.

4.3 Σ -*HCOL* rewriting

Once an *HCOL* expression is translated to the Σ -*HCOL* “normal form,” additional rewriting steps are performed on the Σ -*HCOL* expression to optimize it for efficient code-generation for target architecture. The optimization steps are determined by SPIRAL and validated by HELIX. Each step is implemented in SPIRAL as an application of a “rewriting rule.” Following the translation validation approach discussed earlier, we prove a lemma for each rule and apply them following the trace provided by SPIRAL.

The mechanics of Σ -*HCOL* rewriting are similar to *HCOL* breakdown, described in Section 4.1.2. A few distinctions are described as follows:

4.3.1 Restricted Monoid

Let us consider the following expression which is an `IUnion` (see Section 3.2.10) of a *sparse embedding* (see Section 3.2.9):

$$\mathbb{M}_{n,f,z} \text{ SparseEmbedding}_{a,b,K} \quad (4.1)$$

It can be shown that this operator will produce the same results regardless of the underlying operation f as long as (\mathcal{R}, f, z) forms a *monoid*. This fact could be used to express some rewriting rules in a general way without mentioning the concrete operation in place of f .

However, we can not use $f = \max$ and $z = \text{zero}$ here since the $(\mathcal{R}, \max, \text{zero})$ is not a monoid because zero is neither a left nor a right identity with respect to *max*. On the other hand, for a type $\mathcal{R}^+ \triangleq \{x \in \mathcal{R} \mid \text{zero} \leq x\}$ which is a sub-type of \mathcal{R} , the $(\mathcal{R}^+, \max, \text{zero})$ is indeed a monoid.

Let us consider the Σ -*HCOL* operator from Equation 4.1 in the context:

$$\mathbf{MR}_{n,f,z} \text{ SparseEmbedding}_{a,b,K} \circ (\text{SHPointwise abs}) \quad (4.2)$$

It can be shown that the output of (SHPointwise abs) is actually a vector of values of type \mathcal{R}^+ and in this particular context, our generic rewriting rules, which depend on a monoid, can be used. The difficulty with this approach is that it requires introducing intermediate types and extending all higher-level operators like SHCompose to deal with heterogeneous input and output types. It also requires a more sophisticated setoid rewriting theory to deal with sub-types.

Our alternative approach keeps the types intact and defines the notion of a *Restricted Monoid*. The full definition is shown in Listing 4.6.

```
Require Import MathClasses.interfaces.abstract_algebra.
Require Import MathClasses.theory.setoids.

Section MonoidalRestriction.
  Context A {Ae: Equiv A}.

  (* Predicate on type [A] *)
  Class SgPred A := sg_P: A → Prop.

  (* Restriction of monoid operator and unit values *)
  Class MonRestriction
    {Aop: SgOp A}
    {Aunit: MonUnit A}
    {Apred: SgPred A}: Prop :=
  {
    rmonoid_unit_P: sg_P mon_unit ;
    rmonoid_plus_closed:
```

```

     $\forall a\ b, \text{sg\_P } a \rightarrow \text{sg\_P } b \rightarrow \text{sg\_P } (a \ \& \ b)$ 
  }.

Class RMonoid
  {Aop: SgOp A}
  {Aunit: MonUnit A}
  {Apred: SgPred A} :=
{
  sg_setoid :> Setoid A ;
  mon_restriction :> MonRestriction ;
  rsg_op_proper :> Proper ((=)  $\implies$  (=)  $\implies$  (=)) (&) ;
  rmonoid_ass:  $\forall x\ y\ z,$ 
     $\text{sg\_P } x \rightarrow \text{sg\_P } y \rightarrow \text{sg\_P } z \rightarrow x \ \& \ (y \ \& \ z) = (x \ \& \ y) \ \& \ z ;$ 
  rmonoid_left_id:  $\forall y, \text{sg\_P } y \rightarrow \text{mon\_unit} \ \& \ y = y ;$ 
  rmonoid_right_id:  $\forall x, \text{sg\_P } x \rightarrow x \ \& \ \text{mon\_unit} = x$ 
}.

Class CommutativeRMonoid {Aop Aunit Ares} : Prop :=
{
  comrmonoid_rmon :> @RMonoid Aop Aunit Ares ;
  rcommutativity:  $\forall x\ y, \text{sg\_P } x \rightarrow \text{sg\_P } y \rightarrow x \ \& \ y = y \ \& \ x$ 
}.

End MonoidalRestriction.

```

Listing 4.6: Restricted Monoid

It consists of three typeclasses: *MonRestriction*, *RMonoid*, and *CommutativeRMonoid*, which are parametrized by an abstract type A , a binary function (*SgOp* class instance), and a predicate (*SgPred* class instance). The *MonRestriction* typeclass states that the given “unit” value satisfies the predicate and that the function is closed under the same predicate. The *RMonoid* typeclass extends

MonRestriction with the usual monoid properties: associativity, left identity, and right identity. The *CommutativeRMonoid* extends *RMonoid* with the commutativity property.

Using these classes, we can state and prove that the non-negative values of the \mathcal{R} type form a commutative monoid with max and zero

```
Global Instance NN: SgPred  $\mathcal{R}$  := CarrierAle CarrierAz.

Global Instance RMonoid_max_NN: @RMonoid  $\mathcal{R}$  CarrierAe
  (@max  $\mathcal{R}$  CarrierAle CarrierAleddec) CarrierAz NN. (...)

Global Instance CommutativeRMonoid_max_NN: @CommutativeRMonoid  $\mathcal{R}$  CarrierAe
  (@max  $\mathcal{R}$  CarrierAle CarrierAleddec) CarrierAz NN. (...)
```

The *RMonoid* typeclass can now be used to state lemmas for expressions like those in Equation (4.2). It can also be shown that given an *RMonoid* instance parameterized by a type A , a predicate P , a function f , and a unit value z , we can derive an unrestricted *Monoid* instance for the sigma type $\{x \in A \mid Px\}$ with the following binary function:

```
( $\lambda$  xs ys  $\Rightarrow$ 
  let x := proj1_sig xs in
  let y := proj1_sig ys in
  exist (f x y)
    (rmonoid_plus_closed A x y
      (@proj2_sig A P xs)
      (@proj2_sig A P ys)))
```

and unit value of $(\text{exist } z \text{ (rmonoid_unit_P } _))$.

4.3.2 Example

The result of the Σ -*HCOL* rewriting steps of the expression from Listing 3.7 is shown in Listing 4.7

```

Definition dynwin_SHCOL1 (a:avector 3) : @SHOperator Monoid_RthetaFlags dynwin_i dynwin_o zero
:= SafeCast (SHBinOp Monoid_RthetaSafeFlags (IgnoreIndex2 Zless))
  ◎ Apply2Union Monoid_RthetaFlags plus
    (Embed Monoid_RthetaFlags (le_S (le_n 1))
      ◎ SafeCast
        (IReduction plus
          (SHFamilyOperatorCompose Monoid_RthetaSafeFlags
            (λ jf : {x : ℕ | x < 3},
              SHCompose Monoid_RthetaSafeFlags
                (SHPointwise Monoid_RthetaSafeFlags
                  (Fin1SwapIndex jf (mult_by_nth a)))
                  (SHInductor Monoid_RthetaSafeFlags ('jf) mult 1))
                (Pick Monoid_RthetaSafeFlags (1 + 4) 0))))
        (Embed Monoid_RthetaFlags (le_n 2)
          ◎ SafeCast
            (IReduction max
              (λ jf : {x : ℕ | x < 2},
                SHCompose Monoid_RthetaSafeFlags
                  (SHBinOp Monoid_RthetaSafeFlags
                    (λ (i : {n : ℕ | n < 1}) (a0 b : ℝ),
                      IgnoreIndex abs i
                        (Fin1SwapIndex2 jf (IgnoreIndex2 sub) i a0 b)))
                  (UnSafeCast
                    (ISumUnion
                      (λ jf0 : {x : ℕ | x < 2},
                        Embed Monoid_RthetaFlags (proj2_sig jf0)
                          ◎ Pick Monoid_RthetaFlags
                            (1 + 4) (1 + 'jf * 1 + 'jf0 * (2 * 1))
                        ))))))
    ))))

```

Listing 4.7: Dynamic Window Monitor in Σ -*HCOL* after rewriting

4.4 Σ -*HCOL* to *MHCOL* Translation

4.4.1 Implementation

Translation from Σ -*HCOL* to *MHCOL* is implemented using the Coq meta-programming plugin, Template-Coq [39]. The translation is fairly straightforward, as there is a one-to-one correspondence between language operators, with the exception of Σ -*HCOL* `SafeCast` and `UnsafeCast` operators which are translated as identities. In addition to mapping the operators' names, the translation changes their input types from vectors to memory blocks. The return type of all *MHCOL* operators is a memory block wrapped in an `option` type to facilitate error handling.

Σ -*HCOL* operators are instances of the `SHOperator` typeclass. They can be parameterized by some constants which technically means they can be enclosed in additional lambdas, introducing corresponding variables which could be used inside an operator's shallow embedded definition. For example, our dynamic window monitor Σ -*HCOL* definition from Listing 3.7 is parametrized by parameter a and has the actual type:

`∀ (a: avector 3), @SHOperator Monoid_RthetaFlags (1+(2+2)) 1 zero.`

Such optional parameters will be detected during translation to *MHCOL* and mapped to corresponding binders in the resulting expression. Thus, the result of the *MHCOL* translation of the dynamic window monitor Σ -*HCOL* definition from Listing 3.7 will also be parametrized by a and have the actual type:

`∀ (a: avector 3), @MSHOperator (1 + 4) 1.`

4.4.2 Proof of Semantics Preservation

The semantic equivalence between an Σ -*HCOL* and an *MHCOL* operator is defined as the `SH_MSH_Operator_compat` typeclass. It ensures that they have the same di-

dimensionality and input and output patterns (index sets) and are both structurally correct (by the presence of respective `SHOperator_Facts` and `MSHOperator_Facts` instances). In addition to these properties, it also states the main semantic equivalence property:

```
mem_vec_preservation:
  ∀ (x:svector i),
    (∀ (j: ℕ) (jc: j < i), in_index_set sop (mkFinNat jc) → Is_Val (Vnth x jc)) →
    Some (svector_to_mem_block (op sop x)) = mem_op mop (svector_to_mem_block x)
```

Listing 4.8: Σ -*HCOL* and *MHCOL* main semantic equivalence property

Informally it can be stated as:

For any vector which complies with the input sparsity contract of the Σ -*HCOL* operator, an application of the *MHCOL* operator to such vector, converted to a memory block, must succeed and return a memory block which must be equal to the memory block produced by converting the result of the Σ -*HCOL* operator.

For regular operators, `SH_MSH_Operator_compat` instances can be proven directly. For higher-order operators, the proofs are predicated on `SH_MSH_Operator_compat` assumptions for all operators involved. Some operators may have additional prerequisites. For example, for `Apply2Union`, the output index sets of `f` and `g` must be disjoint.

For translated programs, we use proof automation to prove that `SH_MSH_Operator_compat` holds between the original and the compiled programs.

4.5 *MHCOL* to *DHCOL* Translation

4.5.1 Implementation

Translation from *MHCOL* to *DHCOL* is also implemented in Template-Coq as a recursive descent on *MHCOL*'s AST. The top-level translation function has the following type:

```
Fixpoint compileMSHCOL2DSHCOL
  (res: var_resolver)
  (vars: varbindings)
  (t: term)
  (x_p y_p: PExpr)
: TemplateMonad (varbindings*(DSHOperator)).
```

Since we know that t is Gallina AST of the *MHCOL* operator which is a function with type $(\text{mem_block} \rightarrow \text{option mem_block})$, the compiler is parameterized by two memory pointer expressions specifying where an imperative *DHCOL* program corresponding to this function should read input from (x_p) and write output to (y_p).

The translation, if it succeeds, returns a *DHCOL* program and the list of “global” variables it depends on. Recall that *MHCOL* operators are `MSHOperator` records, which may depend on additional parameters. All such additional parameters will be detected during translation to *DHCOL* and treated as “global” variables which must be present in the evaluation context prior to evaluating the *DHCOL* program. The `varbindings` part of a tuple returned by the compiler is a list of these variable names with their respective types.

The translation procedure is fairly straightforward, as each *MHCOL* operator is compiled to a *DHCOL* fragment, and the translation is compositional. For

example, both `MSHEmbed` and `MSHPick` from *MHCOL* translate to `DSHAssign` in *DHCOL*. Some *MHCOL* operators translate not to a single *DHCOL* operator but rather to a *DHCOL* program fragment. Let us look more closely at an example of how `MSHCompose` is translated to *DHCOL*. The relevant part of the `match` statement on the *MHCOL* operator's name and arguments from AST is shown in Listing 4.9:

```
| Some n_SHCompose, [i1 ; o2 ; o3 ; op1 ; op2] =>
  ni1 ← tmUnquoteTyped N i1 ;;
  no2 ← tmUnquoteTyped N o2 ;;
  no3 ← tmUnquoteTyped N o3 ;;
  (* freshly allocated, inside alloc *)
  let t_i := PVar 0 in
  (* single inc. inside alloc *)
  let x_p' := incrPVar 0 x_p in
  let y_p' := incrPVar 0 y_p in
  let res1 := Fake_var_resolver res 1 in
  '(_, cop2) ← compileMSHCOL2DSHCOL res1 vars op2 x_p' t_i ;;
  '(_, cop1) ← compileMSHCOL2DSHCOL res1 vars op1 t_i y_p' ;;
  tmReturn (vars, DSHAlloc no2 (DSHSeq cop2 cop1))
```

Listing 4.9: `MSHCompose` operator translation to *DHCOL*

Using double square brackets as a shortcut for the `compileMSHCOL2DSHCOL` call, compiling the *MHCOL* operator $\llbracket \text{op1} \circ \text{op2} \rrbracket(x, y)$ will result in the *DHCOL* program `DSHAlloc no2 (DSHSeq $\llbracket \text{op2} \rrbracket(x, t)$ $\llbracket \text{op1} \rrbracket(t, y)$)` which can be summarized with the following pseudo-code:

```
t := allocate o2
t := op2 x
y := op1 t
free t
```

The variables in both *MHCOL* AST and *DHCOL* are referenced by de Bruijn indices. However, since the lexical structure of the two languages is different, a non-trivial mapping between two indices must be established. This is implemented with the help of the “variable resolvers” mechanism, defined in Listing 4.10.

```

Definition var_id := ℕ.
Definition var_resolver := ℕ → var_id.

Definition ID_var_resolver : var_resolver := id.
Definition Fake_var_resolver (parent: var_resolver) (n:ℕ) : var_resolver
  := λ r ⇒ (parent r) + n.
Definition Lambda_var_resolver (parent: var_resolver) (n:ℕ) : var_resolver
  := λ r ⇒ if lt_dec r n then r else (parent (r-n)) + n.

```

Listing 4.10: Variable resolvers

The *var_resolver* is a map from *MHCOL* to *DHCOL* variable indices. The most basic one is `ID_var_resolver` which establishes identity mapping between the two. The next useful resolver is `Fake_var_resolver` which is used when *DHCOL* introduces one or more new variables, which have no counterparts in *MHCOL*. This resolver is stacked on top of an existing resolver modifying its behavior to accommodate for such new variables. This is what is happening in Listing 4.9. We introduce a new `Fake_var_resolver` corresponding to one new variable used to store the allocated temporary memory block. Since this variable is lexically scoped by `DSHAlloc`, the new resolver will be used only while compiling the operators constituting the `DSHAlloc` body. If these operators reference the other variables declared earlier, their references will be computed by adding a unit offset to take into account the new temporary variable introduced by `DSHAlloc`.

In the rest of Listing 4.9, we “unquote”¹ memory block sizes to natural numbers. Then, we compile both operators using the `Fake_var_resolver` with $n = 1$. When compiling two operators, we need to specify the input and output variable indices for each operator. The index of the newly allocated temporary variable will be 0, being the most recently introduced. To reference the original `x_p` and `y_p`, we need to increment their indices by 1 using the `incrPVar` function to take into account the new temporary variable introduced by `DSHAlloc`. This might look similar to what we do in the resolver, but these are *DHCOL* variable indices, which have already been resolved earlier and thus, the resolver mechanism will not be used, and manual adjustment is required. Finally, we construct the resulting expression, which consists of allocation of a temporary memory block followed by sequential execution of the *DHCOL* code corresponding to `op2` and `op1`.

Another resolver we use elsewhere is `Lambda_var_resolver` which is required when compiling functions with n arguments. It is typically used when compiling the function argument of operators like `MSHPointwise` or `MSHBinOp`. It will introduce n new variables with de Bruijn indices $0 \dots n - 1$ representing the parameters of the function. While compiling a function, `Lambda_var_resolver` will map these indices as unchanged. However, when *MHCOL* variables with indices outside this range are mapped, n is first subtracted, the parent resolver is applied, and then n is added back to the result.

To illustrate this, let us consider an *MHCOL* expression which contains a function with one argument $\lambda i.i + a$. We assume the current resolver to be `parent_resolver = Fake_var_resolver (ID_Var_resolver) 1` and the de Bruijn index of a in *MHCOL* is 1. When compiling our lambda function, we use `(Lambda_var_resolver parent_resolver 1)`. When resolving i using this resolver,

¹Template-Coq term which means decoding from AST representation.

we get 0 because of the identity mapping in the lambda arguments. To resolve a with index 1, we first pass $1 - 1$ to the parent resolver which returns 1 as an index of a before entering lambda. Then 1 is added to compensate for the function argument resulting in the final mapping of 2. Variable index mapping for this example is shown in Figure 4.4

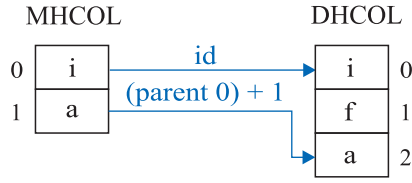


Figure 4.4: Variable resolver example

4.5.2 Proof of Semantics Preservation

While the regular *MHCOL* operators translate to a *DHCOL* program fragment, the higher-order operators translate into a sequence of instructions, with placeholders filled with *DHCOL* translations of their respective parameters. For example, *MHCOL*'s (`MSHIReduction i o n z f op_family`) operator is compiled to the following *DHCOL* program:

```

DSHSeq
  (DSHMemInit o y_p z)
  (DSHAlloc o (DSHLoop n (DSHSeq dop_family (DSHMemMap2 o y_p' (PVar 1) y_p' df))))

```

The parameters of the `MSHIReduction` above are the dimensions of the input and the output vectors (i and o respectively), the size n of the operator family `op_family`, and the initialization value z . The parameters `df` and `dop_family` in *DHCOL* correspond to `f` and `op_family` in *MHCOL*, respectively.

Operators `DSHAlloc` and `DSHLoop` introduce two new variables: the pointer

to a newly allocated memory block and the loop index. Inside the loop, they can be referenced by their respective de Bruijn indices as (PVar 1) and (PVar 0). To evaluate each iteration, the `dop_family` takes the loop index to access the family operator member, which is then executed and writes output to a temporary memory block. The output of `MSHIReduction` is assumed to be written to a memory block referenced by variable `y_p`, and `y_p'` is the same variable with the de Bruijn index increased by two to accommodate for the loop index and a new variable holding a reference to the newly allocated temporary memory block.

We want to prove that our translation from *MHCOL* to *DHCOL* preserves the semantics. As with other HELIX languages, we use an automated *translation validation* approach. To allow automatic proof of translation results, we need to prove correctness lemmas for each *MHCOL* operator and its *DHCOL* translation. Then, these lemmas can be applied recursively, descending the structure of the reified *MHCOL* expression hierarchically.

The first step in the process is to formalize the notion of semantic equivalence between a purely functional language with denotational semantics (*MHCOL*) and an imperative language with operational semantics (*DHCOL*). Each *MHCOL* operator is a function $x \mapsto y$ where `x` and `y` are memory blocks². These functions are *pure functions* without side effects, whose output `y` depends on `x` and other variables in scope. On the other hand, a *DHCOL* translation of this *MHCOL* operator is an imperative program that can read variables available in the *evaluation context* and can also read and modify the memory. One block from this memory will correspond to `x`, and some other block will correspond to `y`. Being a translation of a pure function, the operator can modify only `y`. The formalization of the class of *DHCOL* programs representing pure functions is expressed as a `DSH.pure` typeclass:

²We omit error handling for now.

```

Class DSH_pure (d: DSHOperator) (y: PExpr) := {
  mem_stable: forall  $\sigma$  m m' fuel,
    evalDSHOperator  $\sigma$  d m fuel = Some (inr m') ->
    forall k, mem_block_exists k m <-> mem_block_exists k m';

  mem_write_safe: forall  $\sigma$  m m' fuel,
    evalDSHOperator  $\sigma$  d m fuel = Some (inr m') ->
    (forall y_i , evalPexp  $\sigma$  y = inr y_i ->
      memory_equiv_except m m' y_i)
}.

```

It has the following two properties:

- *memory stability* states that the operator does not free or allocate any memory blocks.
- *memory safety* states that the operator modifies only the memory block referenced by the pointer variable y , which must be valid in the environment, σ .

Now, we can proceed to formulate the semantic equivalence between an *MHCOL* operator and a “pure” *DHCOL* program. Since the *MHCOL* part of this relation is a function, we need to universally quantify on all possible inputs. Since *DHCOL* operators read and modify memory, the input and output of this function must correspond to some existing memory blocks. In *DHCOL* memory, locations can be accessed via pointer variables only, so we state that there are two pointer variables in the evaluation context corresponding to the input and output memory block locations. For convenience, we define semantics equivalence as a type class

parameterized by the respective *MHCOL* and *DHCOL* operators, the evaluation context, and by the name of the input and output pointer variables in this context. Additionally, the purity of the *DHCOL* operator must be guaranteed by providing a `DSH_pure` instance.

```

Class MSH_DSH_compat
  {i o: ℕ} (σ: evalContext) (m: memory)

  (mop: @MSHOperator i o) (dop: DSHOperator)

  (x_p y_p: PExpr) '{DSH_pure dop y_p} :=
  {
    eval_equiv: ∀ (mx mb: mem_block),
      (lookup_Pexp σ m x_p = inr mx) → (lookup_Pexp σ m y_p = inr mb) →
      (h_opt_opterr_c
        (λ md m' ⇒ err_p (λ ma ⇒ SHCOL_DSHCOL_mem_block_equiv mb ma md)
          (lookup_Pexp σ m' y_p))
        (mem_op mop mx)
        (evalDSHOperator σ dop m (estimateFuel dop))));
  }.

```

In the listing above, `h_opt_opterr_c` deals with error handling. While `mem_op` has simple error reporting via option type, `evalDSHOperator` has two-level error handling, distinguishing between running out of fuel and other errors. The equality is defined if both operators err (for whatever reason) or both succeed, in which case, their results must satisfy a provided sub-relation. The sub-relation (expressed via lambda) does additional error handling via `err_p` to ensure that `y_p` lookup succeeds in `m'`. Finally, the equality is reduced to the predicate `SHCOL_DSHCOL_mem_block_equiv` relating memory blocks `mb`, `ma`, and `md`.

Figure 4.5 shows the origin of these values in a case where no errors occur.

Legend: σ is an evaluation context, and m and m' are memory states before and after execution of the `evalDSHOperator`. The `ma` corresponds to a memory block in m' referenced by `y_p`. The `md` is the result of applying the *MHCOL* operator to `mx`.

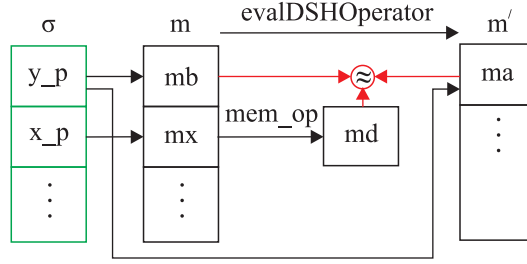


Figure 4.5: *DHCOL* and *MHCOL* equality relation

To understand this relation, we must recall, that in Σ -*HCOL*, sparse vectors represent the results of partial computation. Sparse elements correspond to as yet uncomputed values, while dense elements are already computed. Performing a union of the resulting sparse vectors represents the combining of several partial computations. Replacing immutable vectors with mutable memory blocks allows us to replace the operation of combining computation results with a simple memory update. Following this reasoning, the result of the *MHCOL* operator application (called `md`, where “d” stands for *delta*) is a memory block containing values only at the indices that we need to update. The values at all other indices must remain unchanged. On the other hand, in *DHCOL*, we know the memory state before the operator evaluation and the updated state after it has been evaluated. Thus, `SHCOL_DSHCOL_mem_block_equiv` represents the relation between:

- `mb` - memory state of the output block before *DHCOL* execution
- `ma` - memory state of the output block after *DHCOL* execution
- `md` - values of changed output block elements after *MHCOL* evaluation

This relation is implemented via the element-wise relation, `MemOpDelta`, which is lifted to memory blocks as `SHCOL_DSHCOL_mem_block_equiv`:

```

Definition SHCOL_DSHCOL_mem_block_equiv (mb ma md: mem_block) : Prop
:=  $\forall$  i, MemOpDelta
    (mem_lookup i mb)
    (mem_lookup i ma)
    (mem_lookup i md).

Inductive MemOpDelta (b a d: option CarrierA) : Prop :=
| MemPreserved: is_None d  $\rightarrow$  b = a  $\rightarrow$  MemOpDelta b a d
| MemExpected: is_Some d  $\rightarrow$  a = d  $\rightarrow$  MemOpDelta b a d.

```

Informally, it could be stated as:

For all memory indices in `md` where a value is present, the value at the same index in `ma` should be the same. For indices not set in `md`, the value in `ma` should remain as it was in `mb`.

Once we have proven `SH_MSH_operator_compat` instances for all *MHCOL* operators and their corresponding *DHCOL* equivalents, we can automatically generate proof for the result of any *MHCOL* to *DHCOL* translation as an instance of this class for top-level *MHCOL* and *DHCOL* expressions. During this proof automation, we need to recursively descend on an *MHCOL* expression. The reason for this is that mapping between the two is not injective, and compiling two different *MHCOL* operators could result in similar *DHCOL* constructs not easily distinguishable by simple matching on the structure. Whereas *MHCOL* operators can be uniquely matched.

4.6 *DHCOL* to *FHCOL* Translation

4.6.1 Implementation

Translation from *DHCOL* to *FHCOL* is very straightforward, as both languages belong to the same family and are implemented by different parameterizations of the same module.

Recall that *DHCOL* and *FHCOL* are defined respectively as:

```
Module Export MDSHCOLOnCarrierA := MDSigmaHCOLEval(CarrierAasCT)(MNatAsNT).  
Module Export MDSHCOLOnFloat64 := MDSigmaHCOLITree(MFloat64asCT)(MInt64asNT).
```

Listing 4.11: *DHCOL* module specializations

Translation is implemented as a Gallina function:

```
Fixpoint DSCHOLtoFHCOL (d: MDSHCOLOnCarrierA.DSHOperator) :  
  err MDSHCOLOnFloat64.DSHOperator.
```

Translation works by recursively traversing the structure of `DSHOperator` utilizing one-to-one correspondence between *DHCOL* and *FHCOL* language constructs. However, the translation may fail for one of two reasons:

1. Translation of `CType.t` constants requires converting \mathcal{R} values to `binary64`. Currently, only known constants can be translated, and just two are presently defined, corresponding to `CTypeZero` and `CTypeOne`. Any other constant if encountered will cause translation to fail. The list of known constants could be extended in the future, if needed.
2. Translation of `NType.t` constants requires converting \mathbb{N} values to `Int64.int`. The mapping is not total, as some natural numbers can not fit 64 bits integer representation. The natural numbers which need to be translated appear in

NExpr constants and memory block sizes. If such an out-of-range natural number value is encountered, the translation will fail.

4.6.2 Proof of Semantics Preservation

DHCOL to *FHCOL* translation has not been proven. We expect it to mostly involve numerical analysis. While it is possible to apply generic numeric analysis approaches to try proving the translation for some class of programs and some ranges of input values, this is a case where problem-specific approach may be required, taking into account domain knowledge and specifics of the particular program being compiled (e.g. what external functions it uses, linearity, etc.).

See Sections 5.3.1 and 5.3.2 for discussion of the approaches we devised but have not yet implemented to address this.

4.7 *FHCOL* to LLVM IR Translation

FHCOL programs are further compiled to LLVM IR³ language. This is a low-level language of the LLVM toolchain which can be further compiled to machine code for a variety of supported instruction sets. At the time of this writing, LLVM 3.4 supports code generation for ARM, Qualcomm Hexagon, MIPS, Nvidia Parallel Thread Execution, PowerPC, AMD TeraScale, AMD Graphics Core Next (GCN), SPARC, z/Architecture, x86, x86-64, and XCore. Using LLVM toolchain terminology, our *FHCOL* to IR compiler can be considered an LLVM *front end* for *FHCOL* language⁴.

HELIX relies on the Vellvm project for LLVM IR language formalization and semantics. The compiler produces a deep embedded internal representation of the IR program in Coq. This program can be “pretty printed” (using Vellvm terminology)

³The “IR” stands for *intermediate representation*.

⁴Conversely, from the HELIX point of view, the compiler can be considered a *back end*.

to standard textual representation of IR suitable for further compilation using one of the toolchain’s back-end compilers. Additionally, Vellvm provides ITree-based semantics of LLVM IR [?] based on *interaction trees*[37].

Remembering our goal of proving semantics preservation across the end-to-end chain of HELIX translation steps, we need to connect the semantics of generated IR programs to those of the previous language in the translation chain: *FHCOL*. One way to do this is to prove our compiler correctness by showing that it always preserves the semantics of the *FHCOL* program when it is compiled to IR. This is different from the *translation validation* approach used to prove semantics preservation of translation between other languages in HELIX.

Instead of trying to match *FHCOL* operational semantics directly to LLVM IR ITree-based semantics, we introduce an intermediate step by defining alternative denotation semantics for *FHCOL* and proving that it is equivalent to the *FHCOL* evaluation semantics used in the previous verification step. Then, we prove the compiler’s semantic preservation property based on the denotation semantics of the input language (*FHCOL*) and the ITree-based semantics of the output language (IR). The relation of semantics and languages involved is shown in Figure 4.6. The semantic equivalence relations \approx and \approx_R will be explained in sections that follow.

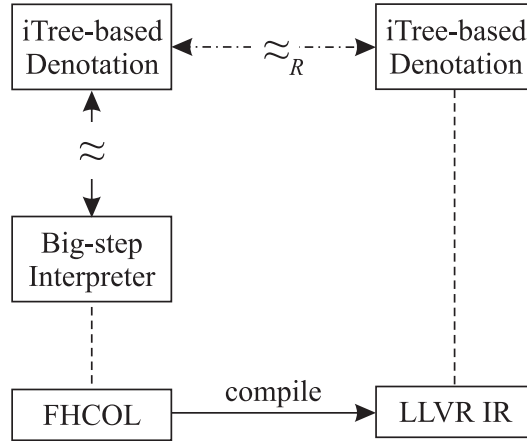


Figure 4.6: *FHCOL* semantics

4.7.1 *FHCOL* to LLVM IR compiler

4.7.1.1 Type Mapping

FHCOL uses `binary64` and `Int64.int` types to represent floating point and integer values, respectively. For floating point values, `binary64` maps directly to LLVM `double` type, as both represent IEEE-754 64-bit binary floats. In Vellvm AST, this type corresponds to `TYPE.Double`.

The *FHCOL* `Int64.int` type is mapped to the IR `i64` type, which in Vellvm AST corresponds to `TYPE.I 64`. The LLVM type system makes no distinction between signed and unsigned integers, unlike C language. IR integers can be interpreted either as signed or unsigned depending on the context. For example, the quotient of two unsigned integers can be computed using `udiv` instruction while for signed integers, `sdiv` instruction must be used. Since *FHCOL* integers are always non-negative, when generating IR code we emit appropriate unsigned instructions.

FHCOL Memory blocks are mapped to IR memory regions and accessed using pointers to `TYPE.Array n TYPE.Double` where n is block size. The `getelementptr`

instruction is used to address individual values of floating point arrays. All new memory blocks allocated by the compiled *FHCOL* program are allocated on the stack frame using `alloca` instruction.

4.7.1.2 Translation Units

A top level translation unit is an *FHCOL* *program* which corresponds to an *FHCOL* operator which may depend on one or more global variables. Programs are described by the record shown in Listing 4.12.

```
Record FSHCOLProgram := mkFSHCOLProgram {
  i o: Int64.int;
  name: string;
  globals: list (string * DSHType);
  op: DSHOperator;
}.
```

Listing 4.12: *FHCOL* Program definition

The record contains the name of the function to generate, a list of global variable names and types, and the `DSHOperator` (from module `MDSHCOLOnFloat64`) which represents the program code. The code of a well-formed *FHCOL* program expects an evaluation environment to be initialized with global variables according to `globals` using the same types and indices, immediately followed by two variables which are pointers to allocated memory blocks of sizes *i* and *o*. The first of these blocks holds the input data vector, and the second one will, upon successful evaluation, hold the result.

All global *FHCOL* variables are mapped to corresponding IR global variables which, depending on the compiler invocation flags, are declared with either the *external* or *internal* linkage type. In the latter case, they are initialized with random

data, as discussed in Section 4.7.2.

4.7.1.3 Compiler Organization

The *FHCOL* to LLVM IR compiler is written in Gallina. It translates *FHCOL* programs into Vellvm AST for the corresponding IR *module*. The main compilation logic is encoded in the `genIR` function, which translates `DSHOperator` into a non-empty sequence of LLVM *blocks*. It takes as a parameter a successor block id, where control should be passed at the end of the compiled operator execution (also known as “destination-passing style”).

```
Fixpoint genIR (fshcol: DSHOperator) (nextblock: block_id) : cerr segment.
```

The top-level compiler entry point used for testing and verification is `compile_w_main`, which performs some additional steps compared to standalone compilation before invoking `genIR`, such as declaring and initializing global variables, generating function declaration for the compiled operator, and generating the *main* function.

There are a few situations where *FHCOL* program compilation may fail. The most obvious example is an invalid *FHCOL* program that attempts to access an undeclared variable or one of the wrong type. Another reason for errors is the integer size restriction: the loop bounds in *FHCOL* operators are expressed as natural numbers which may not fit into the target platform integer size (e.g. `int64`). For *FHCOL* programs generated by HELIX, most of these errors are guaranteed not to occur, but instead of requiring formal guarantees as a pre-condition to invoking the compiler, we add error handling and later prove that the compiler never fails on HELIX-generated programs. The error handling is implemented via the *exception monad*. The compiler returns either a value or an error message as a *string*.

During compilation, the compiler maintains the compiler state. The state consists of integer counters to generate unique LLVM names and the *typing context* Γ . The state data type is shown in Listing 4.13

```
Record IRState := mkIRState {
  block_count: ℕ;
  local_count: ℕ;
  void_count : ℕ;
   $\Gamma$ : list (ident * typ)
}.
```

Listing 4.13: Compiler state frame

Global *FHCOL* variables are mapped to LLVM global variables, while local variables are mapped to registers. Both of their types are recorded in Γ since *FHCOL* does not distinguish between “global” and “local” variables. When compiling the top-level *FHCOL* operator, all pre-defined variables are assumed to be *global* while all new variables created when opening scopes (e.g. memory pointers in `DSHAlloc` or loop indices in `DSHLoop`) are considered *local*.

The counters in `IRState` are used to generate unique identifiers for blocks, local identifiers, and void identifiers. The new identifiers are generated by appending the corresponding counter number to an arbitrary string prefix. The counters are initialized with zeroes and increased after each new variable generation. The combination of counter type and counter value is guaranteed to be unique for each generated identifier. The string prefix is arbitrary and sometimes used to give the identifier a meaningful human readable prefix to make the generated IR code more legible.

Since *FHCOL* is lexically scoped, as the compiler proceeds with structural recursion over the *FHCOL* structure, Γ is used to resolve *FHCOL* variable de Bruijn

indices to compiler-assigned LLVM global variables or register names and their IR types.

The compiler state is maintained in a *state monad*. To combine it with error handling a new monad, combining features of the *state monad* and the *exception monad* is defined with instances for both ExtLib's `MonadExc` and `MonadState` classes.

On several occasions, we need to generate code for loops in the shape described by the following pseudo-code:

```
int i, from, to;

init();
i=from;

while(i<to)
{
    body();
    i++;
}
```

which corresponds to the following IR code:

```
.entry:
    (init)
    %c0 = icmp ult i32 %start, %n
    br i1 %c0, label %.loop, label %.nextblock

.loop:
    %i = phi i32 [ %next_i, .loopcontblock], [ %start, .entry ]
    (body)

.loopcontblock:
    %next_i = add nsw i32 %i, 1
    %c = icmp ult i32 %next_i, %n
    br i1 %c, label %.loop, label nextblock
```

```
nextblock:
```

In particular, the IR code generated for the following *FHCOL* operators makes use of this construct: `DSHMemInit`, `DSHPower`, `DSHIMap`, `DSHBinOp`, `DSHMemMap2`, and `DSHLoop`.

To avoid duplication, the code generation for such loops was generalized via the `genWhileLoop` function:

```
Definition genWhileLoop
  (prefix: string)
  (from to: exp typ)
  (loopvar: raw_id)
  (loopcontblock: block_id)
  (body_entry: block_id) (body_blocks: list (block typ))
  (init_code: (code typ))
  (nextblock: block_id)
: cerr (block_id * list (block typ))
```

The loop initialization code is passed as `init_code`. The loop body is passed as a pre-compiled list of blocks as `body_blocks`. Its entry point is `body_entry`. After execution, it is expected to pass control to `loopcontblock`. To be able to access the loop index variable within the body, its name must be known in advance and thus, it is passed as `loopvar`. Upon completion, the generated loop code will pass control to `nextblock`. The loop bounds are passed as Vellvm expressions `from` and `to`. Finally, the string `prefix` is used to generate new identifiers (using counters from `IRState`). Upon success, `genWhileLoop` returns a list of blocks for the loop and block id of the loop entry point. This loop generalization, in addition to simplifying compiler implementation, is also useful for verification as it allows formulating and proving lemmas about loops in general.

4.7.2 Compiler Correctness

Our goal is to prove *FHCOL* to IR compiler correctness. We define correctness as follows: *For every successfully compiled FHCOL program and the corresponding IR program, assuming that the former does not fail upon evaluation, the latter will not fail either and will compute the same results.*

A few things should be noted about our correctness formulation. First, we only reason about programs which the compiler is able to successfully compile. This is a trade-off. On one hand, we could have tried to prove that an *FHCOL* to LLVM compiler will never fail for programs generated by HELIX's previous steps. Formulating such a lemma would be difficult due to the fact that the previous steps use MetaCoq and we can not reason about them in pure Coq. Alternatively, we could apply an approach where this lemma is automatically generated with MetaCoq and then proven using proof automation. However for practical purposes, its not necessary to prove that the compiler does not fail. The HELIX system user will try to synthesize the code and if this step fails, the user will be presented with an error message. No incorrect program will be produced and no security or safety risks will be posed. This is how other certified compilers, like CompCert, work.

The other interesting feature of our correctness statement is that we only reason about the cases where *FHCOL* evaluation succeeds. It could be shown that it will always succeed for HELIX-generated programs as long as the environment σ is properly initialized with data of expected types. Recall for each *MHCOL* program, we prove `MSH_DSH_compat`, which is our statement of semantic equivalence between *MHCOL* and its *DHCOL* translation. This statement guarantees that the *DHCOL* version will succeed if *MHCOL* does not fail. Furthermore, we have previously proven `MSHOperator_Facts`, which states in part that our *MHCOL* program will not

fail as long as the input memory block respects the operator’s sparsity pattern. These proofs together guarantee that any *DHCOL* program produced by HELIX will not fail on properly initialized input data.

As we recall from Section 4.7.1.2, an *FHCOL* program corresponds to an *FHCOL* operator together with the global variable definitions. To be able to universally quantify compiled programs over all possible values on globals and inputs for verification, we have built a wrapper around `genIR` called `compile_w_main` which, given an *FHCOL* program and some data, produces a self-enclosed IR compilation module, including the following:

1. Declarations of named global variables, initialized with the data provided.
2. Declaration of an IR function corresponding to the compiled *FHCOL* operator. It takes and returns an array of floating-point values.
3. Declaration of a global, anonymous variable which acts as a placeholder for the input data. It is statically initialized with the data provided.
4. Declaration of a global, anonymous variable which acts as a placeholder for the results of computations performed by the *FHCOL* program. Upon completion of the generated IR program, the result of computation performed by the *FHCOL* operator on the given input data will be stored here.
5. Declaration of the `main` function which calls the above-mentioned function with the data from the input placeholder variable and then loads and returns data from the global output placeholder variable.

The structure of the generated code is illustrated in Listing 4.14.

```
#define I_SIZE ... /* input size */
```

```

#define O_SIZE ... /* output size */

float global1 = ...;
float[] global2 = {...};

float[I_SIZE] _X = {...};
float[O_SIZE] _Y = {...};

void f(float *x, float *y)
{
    /* code, generated from FHCOL */
}

float[O_SIZE] main() {
    f(_X, _Y);
    return(*_Y);
}

```

Listing 4.14: IR module organization in pseudo-C

If it succeeds, `compile_w_main` returns a self-contained *IR module* which is specialized with the given values of the global variables and the input vector. It has an entry point called `main` which returns the result of executing the compiled *FHCOL* program.

To obtain denotation of the *FHCOL* program in `denote_FSHCOL`, we first perform initialization steps which generate events for allocation of global variables, input vectors, and output vectors. Memory blocks for globals and for the input vector are initialized with data provided to `denote_FSHCOL`. Then after adding events produced by `denoteDSHOperator`, an additional event is generated, which loads the output vector from memory, similar to the `return` statement in the `main` function

in Listing 4.14.

Showing that for all *FHCOL* programs p and for all possible data, the itrees which are produced by denotating `compile_w_main p data` results and `denote_FSHCOL p data`, when interpreted, return the same value will provide basic compiler correctness. It will not guarantee a specific state of the memory or the environment upon completion of the program. Additional safety properties, like this and upper bounds on allocated memory, could be proven later. LLVM specification also defines *undefined behaviors*, *poison* values, and *undef* values. HELIX-generated programs do not expose any of these features and thus, they do not affect our correctness guarantees or proofs. The top-level compiler correctness lemma is shown in Listing 4.15.

```
Theorem compiler_correct:
  ∀ (p: FSHCOLProgram)
    (data: list binary64)
    (pll: toplevel_entities typ (LLVMast.block typ * list (LLVMast.block typ))),
  ∀ s, compile_w_main p data newState ≡ inr (s,pll) →
    eutt (bisim_final []) (semantics_FSHCOL p data) (semantics_llvm pll).
```

Listing 4.15: IR compiler correctness theorem

The equality between two itrees is expressed via the `eutt` relation (also known as “equivalence up to taus”) parameterized by the `bisim_final` relation. This corresponds to \approx_R in Figure 4.6. More specifically, $t_{\text{helix}} \approx_{\text{bisim_final}} t_{\text{llvm}}$ relates t_{helix} and t_{llvm} itrees if they are weakly bisimilar (i.e. they produce the same tree of visible events, ignoring any finite number⁵ of silent *Tau* steps) where all values returned along corresponding branches are related by `bisim_final`.

The `bisim_final` relation compares the return value of the `main` function in

⁵Finite number of *Tau* steps is a condition of non-divergence.

LLVM with the evaluation results of the *DHCOL* program. Even though the final relation does not examine memory or environment, to prove the above-mentioned top-level compiler correctness theorem, we need to reason not only about return values but also about program states during each step of execution. There are three main stages of execution:

1. Initialization of global variables.
2. Execution of `main` function.
3. Execution of function which represents compiled *FHCOL* operator.

The main effort lies in proving the correctness of the compilation of the *FHCOL* program, which consists mostly of reasoning about `genIR` and `denoteDSHOperator` functions. Both of them are implemented via structural recursion on `denoteDSHOperator`, and the proof can be expressed as an `eutt` equality using an invariant of the compiler and the interpreter states. The main invariant carried along combines the following three properties:

1. The memory state satisfies the *memory invariant* (discussed below).
2. The environment state is well-formed; the σ typechecks with respect to Γ .
3. The counters for generating unique LLVM names are consistent with respect to the current environment, i.e. they indeed contain a high-water mark for each type of identifier, which will ensure that the newly generated names will never collide with existing variables.

The memory invariant relies on HELIX's evaluation context σ and the typing context Γ built by the compiler. It states that for any variable in σ named x and its corresponding type τ from Γ , the value is correct according to the following rules:

1. For τ in *integers* and *floats*, there is a matching variable named x in either the global or local IR environment. For local variables, the scalar `uvalue` in the environment should match the value from σ . The global variables, by HELIX convention, are always pointers. They must point to the valid memory location which holds the corresponding `uvalue`.
2. For τ corresponding to pointers, HELIX pointers have a matching pointer in the local or global IR environment, and they both point to identical arrays.

FHCOL to LLVM IR compiler correctness proof is close to completion. Working in collaboration with UPenn’s Vellvm team, we have already proven correctness for all arithmetic expressions (`AExpr`, `NExpr`) and higher-order *FHCOL* operators, such as `DSHLoop` and `DSHSeq`, memory allocation (`DSHAlloc`), and assignments (`DSHAssign`). While proving these operators, we identified and formally defined required invariants and developed proof techniques along with a supporting library of lemmas and Ltac scripts, which will be used to conclude compiler correctness proofs. We do not foresee any significant challenges proving the remaining operators. A few bugs in the compiler were identified and fixed in the process. We plan to publish a separate paper describing our compiler correctness proofs.

4.7.3 Compiler Testing

To test our *FHCOL* to LLVM IR compiler during development before formally verifying it, we developed a testing framework and defined several compiler tests. These tests were run either manually or as a part of HELIX continuous integration. Each test was defined in Gallina as an `FSHCOLProgram` record which contains an *FHCOL* program; its input and output vector dimensions; and a list of the global variable names and types it depends upon. The *FHCOL* operator in the test is

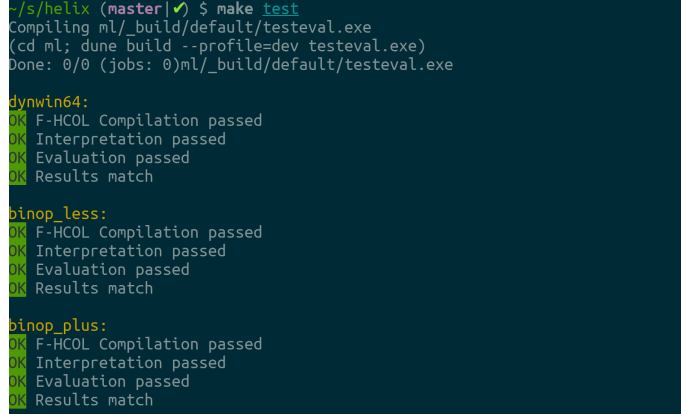
supposed to be produced by HELIX and must follow all of its conventions on variable naming, sparsity, etc., even though this is not enforced by the testing framework.

A single test execution for a given `FSHCOLProgram` performs the following steps:

1. Initialize a data buffer with some random floating-point values. This buffer will be used as a data pool to initialize all global variables and the input memory array during the test. The required data pool size is computed based on the types of global variables and the input vector size from `FSHCOLProgram`. As an additional precaution, the buffer will be treated as cyclic, so if the size is insufficient, some of the random values will be re-used.
2. Compile *FHCOL* program to LLVM IR, represented as in-memory AST. The program contains a self-enclosed IR module with global variables and an input array initialized with data from the provided data pool, as discussed in Section [4.7.2](#).
3. Run Vellvm interpreter on the generated IR program, executing the `main` function and recording the value it returns.
4. Run the *FHCOL* big-step evaluator giving it sufficient fuel (as estimated by `estimateFuel`) and an initial memory state with σ initialized with global variables, an input vector holding the data taken from the data pool, and the placeholder for output data. Upon successful evaluation, note the contents of the memory block designated to hold the result of computation.
5. Compare results of successful *FHCOL* evaluation and IR interpretation steps.

Most of the testing logic, like running the evaluator and interpreter, is implemented in Gallina and extracted to OCaml. Native OCaml code is used to generate

random numbers, call extracted test drivers, and to compare the results. For a test to pass, all steps above must succeed. For the test suite to pass, all tests must succeed. A partial screenshot of a successful run of the test suite is shown in Figure 4.7.



```

~/s/helix (master) $ make test
Compiling ml/_build/default/testeval.exe
(cd ml; dune build --profile=dev testeval.exe)
Done: 0/0 (jobs: 0)ml/_build/default/testeval.exe

dynwin64:
OK F-HCOL Compilation passed
OK Interpretation passed
OK Evaluation passed
OK Results match

binop_less:
OK F-HCOL Compilation passed
OK Interpretation passed
OK Evaluation passed
OK Results match

binop_plus:
OK F-HCOL Compilation passed
OK Interpretation passed
OK Evaluation passed
OK Results match

```

Figure 4.7: A screenshot of the compiler test suite execution

We’ve defined eleven tests with different levels of complexity. Some test a single *FHCOL* operator, while more complex ones compute the *Chebyshev distance* or calculate the dynamic window monitor expression. The latter, being the most complex test in the suite, produces 368 lines of IR code shown in Appendix B.1.

4.7.4 DHCOL Denotation Semantics

We provide an alternative *DHCOL* semantics based on interaction trees. The type of events describing HELIX interactions with its environment is defined as a monad:

```

Variant MemEvent: Type → Type :=
  | MemLU (msg: string) (id: mem_block_id): MemEvent mem_block
  | MemSet (id: mem_block_id) (bk: mem_block): MemEvent unit
  | MemAlloc (size: NT.t): MemEvent mem_block_id
  | MemFree (id: mem_block_id): MemEvent unit.

```

Listing 4.16: DHCOL events

It reflects the fact that the only environment *DHCOL* program modifies is the memory, as defined by the memory model. The events describe the operations of allocation and de-allocation of memory blocks as well as reading and writing them.

Additionally, we distinguish two types of failures: *static* and *dynamic*. The former refers to errors indicating an ill-formed program which could be detected at the compilation stage. An example of such an error is access to an undefined variable or a loop bound which can not fit the platform integer size. In the presence of such errors, *DHCOL* compilation will fail. In contrast, *dynamic errors* refer to errors that can not cause compilation failure but do cause the evaluation to fail. The denotation of a *DHCOL* program can fail with either static or dynamic failures, although this distinction is not currently used, because our compiler correctness proof is stated only for programs which evaluate successfully. However, this mechanism will allow us to extend the correctness formulation in future to reason about failing programs as well. For example, we can try to prove that evaluation and compilation would fail with the same errors, which could be a useful security property for a compiler. Both failures are implemented as exception monads and, combined with HELIX memory events, represent a complete model of the *DHCOL* interaction with its environment, including error handling:

```
Definition StaticFailE := exceptE string.  
Definition DynamicFailE := exceptE string.  
Definition Event := MemEvent +' StaticFailE +' DynamicFailE.
```

To be able to interpret memory events, we define an event handler which maps them to the state monad on HELIX memory extended with dynamic and

static failures:

```

Definition Mem_handler:
  MemEvent ~> Monads.stateT memory (itree (StaticFailE +' DynamicFailE))
:=  $\lambda$  T e mem  $\Rightarrow$ 
  match e with
  | MemLU msg id  $\Rightarrow$ 
    lift_Derr (Functor.fmap ( $\lambda$  x  $\Rightarrow$  (mem,x)) (memory_lookup_err msg mem id))
  | MemSet id blk  $\Rightarrow$  ret (memory_set mem id blk, tt)
  | MemAlloc size  $\Rightarrow$  ret (mem, memory_next_key mem)
  | MemFree id  $\Rightarrow$  ret (memory_remove mem id, tt)
  end.

```

The `lift_Derr` function converts failed memory lookups into dynamic errors.

The memory interpretation can only produce dynamic errors.

Similar to compilation and evaluation, the denotation is implemented via structural recursion. The fixpoint definition does not return any useful value but produces an event tree:

```

Fixpoint denoteDSHOperator ( $\sigma$ : evalContext) (op: DSHOperator): itree Event unit.

```

The high-level theorem of semantic equivalence between *DHCOL* evaluation and denotation semantics corresponding to \approx in Figure 4.6 is stated as:

```

Theorem Denote_Eval_Equiv_Succeeds:
   $\forall$  ( $\sigma$ : evalContext) (op: DSHOperator) (mem mem': memory) (fuel:  $\mathbb{N}$ ),
    evalDSHOperator  $\sigma$  op mem fuel = Some (inr mem')  $\rightarrow$ 
    eutt eq (interp_Mem (denoteDSHOperator  $\sigma$  op) mem) (ret (mem', tt)).

```

Listing 4.17: DHCOL semantic equivalence

It says that if the evaluation of the *DHCOL* operator succeeds, then the resulting memory will be equivalent to the memory produced by interpreting the

corresponding itree (assuming that both start from the same initial memory state).

The denotation and evaluation semantics and their equivalence are defined for the *DHCOL* family of languages and apply to *FHCOL*, which is one member of that family.

4.7.5 Example

The final result of the compilation of the Dynamic Window Monitor to LLVM IR using HELIX is shown in Appendix [B.1](#).

Chapter 5

Results and Discussion

5.1 Implementation

The current size of HELIX development is 45 KLOC of Coq. The proof to specifications ratio is approximately 2:1. There is an insignificant amount of OCaml code (less than 500 lines) implementing the test harness.

Module	loc	Percent
<i>HCOL</i>	1,200	2.65%
Σ - <i>HCOL</i>	11,044	24.40%
<i>MHCOL</i>	8,074	17.84%
<i>DHCOL</i>	10,820	23.91%
<i>FHCOL</i>	513	1.13%
LLVM	7,291	16.11%
Tactics	87	0.19%
Util	4,928	10.89%
Dynamic Window Monitor	1,304	2.88%
TOTAL	45,261	100.00%

Table 5.1: Implementation code size

The distribution of code in the lines of code (loc) between modules is shown

in Table 5.1. Modules corresponding to HELIX languages (e.g. *HCOL*, Σ -*HCOL*) include both formalization of the given language as well as translation with the proof from the previous language in the chain. For example, the Σ -*HCOL* module includes: 1) Σ -*HCOL* language formalization; 2) *HCOL* to Σ -*HCOL* compiler implementation and proofs; and 3) Σ -*HCOL* rewriting engine implementation and proofs. The *Dynamic Window Monitor* module includes a complete dynamic window monitor example, first defined in *HCOL* and then translated through all intermediate steps to LLVM IR with end-to-end proofs.

HELIX is a rather sizable Coq project, and we encountered various engineering challenges while working on it.

Coq ecosystem maturity was an ongoing concern. We started with Coq version 8.4 and at the time of writing, we were on Coq version 8.12. We encountered and reported several bugs in Coq and the standard library. Coq documentation was lacking in many places, and we had to resort to the help of mailing lists, forums, and personal connections to fill the gaps. Luckily, the Coq community was very helpful and responsive. The 3rd party Coq libraries that we started using, such as *ExtLib*, *MathClasses*, *CoLoR*, *Vellvm*, and *MetaCoq* introduced additional dependencies and maintenance challenges. Oftentimes, there was a delay in supporting the latest Coq versions by these libraries, and we had to wait for maintainers to update them before we could switch to the new versions of Coq. We encountered some bugs in these libraries which we reported and in some cases contributed patches. The authors of these 3rd party projects and the Coq community were very helpful in resolving these issues. We experienced Coq ecosystem adopting tools, like *OPAM* package manager and *dune* build system, which contributed greatly to the robustness of our project infrastructure. Coq itself recently made great progress towards becoming a reliable,

well-supported project. The Coq team has adopted a regular, predictable release cycle and has improved documentation and packaging, among other enhancements. The ecosystem of 3rd party libraries and packages is still a mixed bag. Some of them are well supported and maintained, while others are more academic, showing interesting results and techniques but not intended or suitable for use in 3rd party developments.

Performance was another area of concern, starting with build performance. As of this writing, it takes about 30 minutes to compile HELIX on a modern laptop or desktop computer (*Intel Core i7*, quad-core, 16GB RAM, SSD), but this is only after some code optimizations as well as recent improvements in Coq performance. Before that, we had to deal with builds of one hour or more. We observed that the build process does not always utilize all CPU resources available. One reason for that is the fact that the Coq compiler does not use multiple CPU cores. This is an inherent restriction of the current OCaml compiler implementation, and it is currently being actively worked on by the OCaml community[50]. Meanwhile, the only opportunity to use multiple CPU cores in Coq builds is via parallel build, compiling several source files in parallel. This works to the point where dependencies between modules prevent such parallelism. One part of the Coq system which is responsible for a significant share of our performance problems is *typeclass resolution*. We actively use typeclasses in HELIX formalization and proofs. Coq typeclass search could be very slow. It could be guided towards faster resolution with the *Hints* mechanism, but it is rather blunt and does not allow fine control over the search. In an attempt to diagnose search performance bottlenecks, we wrote a tool to parse the debug output of a typeclass search and plot it as a graph. Finally, we embarked upon a project to enhance Coq typeclass search by introducing a caching mechanism. This was

implemented as a patch to Coq OCaml sources. In our implementation, we cache failures of typeclass resolution. The cache is a set, and we check for membership. It is invalidated by changes in hint databases or other variables affecting the typeclass resolution mechanism. Comparing goals is computationally heavy, so we need to strike a balance between the cost of cache lookups and the number of cache hits. Both addition and lookup of the new goals to the cache are heavy operations $\mathcal{O}(n)$. Our initial naive implementation (*strict match*) gave us a 2.49% hit rate with a max cache size of approximately 24,000 entries. Implementing a more intelligent match up to unresolved *evars* (*evvar match*) increased the hit ratio to 14%, and the max cache size became more manageable: below 4,000. Our final observation was that due to the cost of cache lookup, it does not make sense to check the cache for goals which are the leaves in the proof search tree. We introduced a `min_goals` parameter which controls how many dependent goals a goal must have to be included in caching. This slightly decreased the cache size and marginally improved the performance. Our caching implementation compiling *math-classes* Coq library, which makes heavy use of type classes, has shown 27% speed up. The patch was submitted¹ to Coq maintainers but was not accepted in favor of a future, as yet unspecified, more general solution to this problem.

5.2 Coverage

HELIX is a prototype system and as such has some limitations. It could be extended in future and developed further to become a production-level counterpart of SPIRAL. A fair amount of engineering work is required to make it such. However, it is an entirely feasible engineering project, and HELIX can eventually be devel-

¹See <https://github.com/coq/coq/issues/6213>.

oped into a code-generation system for general practical use. A few limitations and shortcomings of the current implementation are worth mentioning:

SPIRAL integration. HELIX uses intermediate OL and Σ - OL expressions as well as a sequence of rule applications computed by SPIRAL. Our original plan was to parse the SPIRAL log file to extract the sequence of rules, map them to corresponding HELIX lemmas, and use them to automatically generate a proof. Due to time constraints, we have not implemented SPIRAL log file parsing. For our running example and tests, we have manually extracted and applied these rules. Similarly, SPIRAL-generated intermediate OL and Σ - OL expressions were manually translated to $HCOL$ and Σ - $HCOL$, respectively. This translation could be trivially automated, as it is purely syntactic.

Numerical analysis. $DHCOL$ to $FHCOL$ translation has not been proven. It mostly involves numerical analysis. See Sections 5.3.1 and 5.3.2 for discussion of the approaches we devised but have yet to implement.

Proof automation. Current proof automation is relatively basic and could be improved.

Refactoring, optimization, and cleanup. A list of smaller improvement and code cleanup opportunities is listed in the `TODD.org` file in the HELIX git repository.

5.3 Future Research Directions

Besides the implementation shortcomings, listed in the previous section, we identified several interesting future research directions, discussed below.

5.3.1 Integer overflow proofs

When translating from *DHCOL* to *FHCOL*, we switch from natural numbers to fixed-length integers for loop indices and offsets. The out-of-bounds check for all constants is performed during translation, as discussed in Section 4.6. However during execution as arithmetic computations are performed to calculate offsets, integer overflow could theoretically occur. We need to perform some numeric analysis to prove this never happens. This should be fairly straightforward, for the following reasons:

1. All arithmetic expressions involve only loop indices and constants. No input data is used.
2. Loop indices are bounded by constants.
3. The language does not support recursion, and the only construct we should be concerned with during integer overflow analysis is a (potentially nested) loop.
4. The integer arithmetic of `NExpr` is relatively simple and well defined for both natural numbers and fixed-length integers.

One practical consideration worth mentioning is that *HELIX* is designed to operate on dense vectors in memory. Consequently, for most of the practical applications we envision, the individual vectors typically have relatively small sizes. Since generated loop bounds are usually linked to data dimensions, that also makes them relatively small, and the overflow is unlikely to occur.

5.3.2 Floating-point proofs

Floating point numbers present a distinct set of challenges. Instead of introducing floating point numbers, we work on an abstract data type (generalized reals) up to

and including the *DHCOL* language. The next language in the chain is *FHCOL*, which replaces the abstract data type with IEEE floating point numbers. The relation in the real domain between the results of the evaluation of the structurally similar expressions in these two languages could encapsulate all numeric analysis properties, such as error bounds and numerical instability. We identified three approaches:

Offline Uncertainty Propagation. In the most general case, an uncertainty propagation approach can be applied [51]. Using it, we can estimate error bounds for compiled *FHCOL* expressions. Unfortunately, in many cases, the resulting error bounds are too large to be useful for practical applications.

Problem-specific Error Estimation. Since our system is intended to validate cyber-physical systems, it is likely that the problem domain could inform additional physical constraints which would allow us to provide stronger guarantees, such as tighter error bounds. Therefore, instead of trying to solve this problem in general, we allow users to plug in their own reasoning here, by providing a lemma which will guarantee that for a particular expression and its value ranges, the floating point approximation meets the user’s given criteria.

Online Uncertainty Propagation. This approach is similar to the *Abstract Interpretation* with the *interval domain*, where the values are represented as intervals which are computed at runtime. The result of the computation is not a single value but an interval. Because it is computed for concrete values, it is usually much narrower than one estimated using offline uncertainty propagation. In some cases, it could be proven that it is smaller than the *machine epsilon* of the floating point

representation, collapsing the output interval into a single floating point number. The price of this approach is that additional computations have to be performed at runtime affecting the performance. The unique advantage of HELIX here is that such computations could be compiled into highly efficient parallelized and vectorized code using SPIRAL’s optimizations. For this approach, we will introduce yet another language named *IHCOL* in which we will represent an \mathcal{R} value as an interval bounded by two IEEE floating point numbers.

5.3.3 Finite-sets proofs

In automated proofs when applying per-rule lemmas, sub-goals for additional pre-conditions may be generated. These are mostly either arithmetic goals related to index bounds or goals related to (non)intersection, equality, or inclusion of sparsity patterns represented as finite sets. The former are easily solved automatically by tactics like `lia`. The latter are solved manually at present. Recall that we use type `Ensemble (FinNat n)` to represent a set of finite numbers bounded by $n \in \mathbb{N}$. The set membership is decidable and typically, n is relatively small as it represents vector size. An example proof obligation which needs to be solved manually after HELIX proof automation is shown in Listing 5.1:

```

∀ (a: vector  $\mathcal{R}$  3) (j :  $\mathbb{N}$ ) (jc : j < 2),
  Included (FinNat 2) (Full_set (FinNat 2))
    (Union (FinNat 2) (singleton 1)
      (Union (FinNat 2) (singleton 0) (Empty_set (FinNat 2))))

```

Listing 5.1: Finite set proof obligation example

It is relatively simple, and the solving of this type of sub-goal can be automated. One approach would be through classic Coq automation using *Ltac*. How-

ever, a more interesting approach is to try to use *computation reflection* [52].

5.3.4 LLVM vector instructions

To generate high-performance code, SPIRAL uses C compiler intrinsics to generate SIMD instructions. In LLVM IR language, they correspond to using vector arguments in arithmetic operators. For example, a `fadd` operator could take two arguments of type `<4 x double>` to perform pointwise addition of two vectors of four double-precision floating point values. When IR is compiled to machine code for a platform supporting SIMD instructions (e.g. Intel SSE, AMD 3DNow!, Motorola AltiVec, or IBM SPU), `fadd` could be compiled to a single CPU instruction which performs such addition, while for platforms without SIMD support, it will be translated to a sequence of four scalar additions.

HELIX does not currently use IR vector arithmetic as it is not yet fully supported by the formal IR semantics of the Vellvm project. Once support for these instructions is added, HELIX can be modified to generate even more efficient code.

5.3.5 Algebraic Theory of Partial Computations

Our approach where partial computations are represented as operations on sparse vectors presents an interesting, long-term research direction and could potentially be developed into a comprehensive algebraic theory of partial computations.

5.4 Related Work

Encoding program transformations as rewriting rules using algebraic properties of the language is a well established approach in the field (see for example, [53], [54], [55]). These systems use functional program rewriting without the use of our sparse

vector parallelism abstraction, and none of them are formally verified.

Another attempt to formally verify SPIRAL is described in [56]. In this work, the author limits himself to a subset of SPIRAL *OL* language where all operators are linear and thus could be evaluated to matrices. This informs his linear algebra approach to operator equivalence and rule validation. He also stops one step short of actual machine code generation (directly or via an intermediate language like CompCert *Clight*) with the output language to which SPIRAL expressions are compiled being a simple imperative language with arrays called IMP+V. The formalization and proofs related to Σ -*OL* are listed as future work. As this project is also implemented in Coq, it might be possible to combine it with our work, re-using his matrix factorization theory for linear operators and related rules in HELIX.

There are several projects for certified compilation from functional to imperative languages. *CertiCoq* translates Gallina programs to CompCert’s *Clight*. The goal is much more ambitious than ours, as the aim is to translate not a domain specific language like Σ -*HCOL* but a dependently typed general purpose language. Interestingly, all three guiding principles cited in [57] also apply to our approach. Some of their transformation steps could be related to ours. For example, going from dependently typed Σ -*HCOL* to *MHCOL*, we perform nominal *type erasure*. However, there are some differences at later stages. For example, unlike HELIX, *CertiCoq* uses a continuation-passing style representation and proves compiler correctness, while HELIX relies on automated translation validation.

Another related project is *CakeML* [58], which also targets a subset of a general-purpose language (Standard ML). Unlike HELIX, *CakeML* uses higher-order logic (HOL) to specify *functional big-step semantics* [59]. There are similarities with our approach, as we also use a *definitional interpreter* [60] written in Gallina, to de-

fine the semantics of the higher-order language, *FHCOL*. The main differences are: small-step versus big-step semantics and translation validation versus certified compilation. Additionally, our programs can not diverge, and while we technically use *fuel* to simplify termination checking, this differs from the *clock* usage in CakeML.

5.5 Contributions and Lessons Learned

Below is a quick summary of key contributions and results of this work.

1. Formal methods
 - (a) Formalizing a class of Operator Languages.
 - (b) Reasoning about partial computations algebraically using sparse vectors.
 - (c) Dual semantics approach for building a certified compiler.
 - (d) Several approaches for dealing with floats.
2. System
 - (a) End-to-end system prototype (with multiple deep and shallow embedded languages using translation validation and certified compilation).
 - (b) LLVM-based, verified code generation backend for SPIRAL.
 - (c) Formal verification of existing system using translation validation.
3. Coq
 - (a) Switching between deep and shallow embedding.
 - (b) Monadic sparsity.
 - (c) Automating translation validation.

- (d) Using typeclasses to formalize properties of operators.

The first lesson we learned from HELIX development is that formal verification is still a hard problem. While core tools are available and some techniques are developed it is still a laborious process, requiring a lot of tedious proving. It tells us that the field is still far from maturity and there are many opportunities to make formal verification more streamlined and easier to estimate engineering process.

Another outtake is that *translation validation* is a very powerful approach which could be used instead of heavier full compiler verification. The main argument against it, is usually that it requires additional manual proof work for each program. We have shown that for some languages these proofs could be automated, making it suitable for developing self-contained, fully automated certified compilers.

5.6 Concluding Remarks

The diversity and complexity of computer hardware is likely to continue to grow, making manual implementation of high-performance numeric algorithms more and more challenging. The optimization space required to generate high-performance code will be grow exponentially with the number of hardware capabilities. Even now, a good implementation already needs to take into account the heterogeneous use of TPUs, GPUs, and CPUs, instruction timing, instruction cache behavior, SIMD instruction use, the number of CPU threads and cores, the number of registers, memory access speed, etc.

It is our belief that the future direction of numerical computing is heading towards high-level, possibly declarative, languages used to describe numeric algorithms. These languages could be equipped with strong type systems and formal semantics and compiled into efficient code for various target hardware platforms.

Compilation could involve going through several intermediate languages, each gradually decreasing the level of abstraction towards the target hardware architecture. During such compilation (or rather optimal code synthesis), the performance-related code transformations could be performed at every step but as early as possible. Each level of these transformations will be backed up by sound theory (algebraic, functional, or computational) which will allow automatic generation of proofs of correctness of the generated code. This is the approach HELIX demonstrates.

Appendix A

DHCOL Big-Step Operational Semantics

In this appendix, we present Big-Step Operational Semantics for the *DHCOL* language, informally described in Section 3.4. This semantics is implemented as a fix-point evaluator, described in Section 3.4.3. The *DHCOL* language syntax is made up of syntactic sets, which are listed below along with their respective Coq types:

1. Natural numbers (\mathbb{N} type)
2. Unsigned fixed-length machine integers (`NT.t` type)
3. Values of abstract *carrier type* (`CT.t` type)
4. Constants zero and one (`CT.t` type)
5. Strings (`string` type)
6. Memory blocks (`mem_block` type)
7. Constant `mem_empty` for an empty memory block (`mem_block` type)
8. Expressions (`NExpr`, `PExpr`, `MExpr`, and `AExpr` types)
9. Operators (`DSHOperator` type)

In presenting *DHCOL* syntax, we will use the following conventions:

- lower-case names with subscript $_s$ will range over **string**
- lower-case names with subscript $_N$ will range over natural numbers in \mathbb{N}
- lower-case names with subscript $_i$ will range over unsigned fixed-length machine integers in **NT.t**
- lower-case names with subscript $_c$ will range over abstract *carrier type* values in **CT.t**
- lower-case names with subscript $_b$ will range over memory blocks in **mem_block**
- lower-case names with subscript $_n$ will range over **NExpr**
- lower-case names with subscript $_p$ will range over **PExpr**
- lower-case names with subscript $_m$ will range over **MExpr**
- lower-case names with subscript $_a$ will range over **AExpr**
- upper-case names without subscript will range over **DSHOperator**

A.1 The Set of States

The set of states has type $\text{evalContext} \times \text{memory}$, usually denoted as (σ, m) . The **evalContext** is a list of variable types and bindings of type **DSHVal**:

DSHVal := **DSHnatVal** value_i | **DSHTypeVal** value_c | **DSHPtrVal** value_N size_i

We use square bracket notation $\sigma[\text{index}_N]$ to reference a variable with the de Bruijn index index_N in the evaluation context σ . The type of $\sigma[\text{index}_N]$ is **err DSHVal**. It will be evaluated to **inr value** in case of successful lookup or **inl _** if there is no variable with such an index in σ .

The **memory** represents the state of the memory, per the memory model described in Section 3.3.1. We also use square bracket notation $m[\text{addr}_N]$ to reference a memory block at address addr_N . Thus, the type of $m[\text{addr}_N]$ is **err mem_block**.

It will be evaluated to `inr valueb` in case of successful lookup or `inl _` if the address is out of range.

Finally, we write $block_b[\text{offset}_\mathbb{N}]$ to access a value at offset $\text{offset}_\mathbb{N}$ in memory block $block_b$. The type of such expression is `err CT.t`. It will be evaluated to `inr valuec` in case of successful lookup or `inl _` if the offset is out of range.

A.2 Expressions

Expressions with types `PExpr`, `MExpr`, `NExpr`, and `AExpr` are evaluated in a given state (σ, m) to their respective values of type $(\mathbb{N} \times \text{NT.t})$, $(\text{mem_block} \times \text{NT.t})$, `NT.t`, and `CT.t`. The evaluation can fail, so the `err` monad is used to wrap the evaluation results. The relation between an expression together with the state and the evaluation result is denoted as \Downarrow .

PExpr evaluation semantics

$$\frac{\sigma[x_\mathbb{N}] = \text{inr } (\text{DSHPtrVal } v_\mathbb{N} \text{ size}_i)}{\langle \text{PVar } x_\mathbb{N}, \sigma, m \rangle \Downarrow \text{inr } (v_\mathbb{N}, \text{size}_i)} \text{PVarOK}$$

$$\frac{\sigma[x_\mathbb{N}] = \text{inl msg}_s}{\langle \text{PVar } x_\mathbb{N}, \sigma, m \rangle \Downarrow \text{inl "error looking up PVar"}} \text{PVarErr}$$

MExpr evaluation semantics

$$\frac{\langle x_p, \sigma, m \rangle \Downarrow \text{inr } (x_\mathbb{N}, \text{size}_i) \quad m[x_\mathbb{N}] = \text{inr } v_b}{\langle \text{MPtrDeref } x_p, \sigma, m \rangle \Downarrow \text{inr } (v_b, \text{size}_i)} \text{MPtrDerefOK}$$

$$\frac{\langle x_p, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{MPtrDeref } x_p, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{MPtrDerefErr1}$$

$$\frac{\langle x_p, \sigma, m \rangle \Downarrow \text{inr } (x_{\mathbb{N}}, \text{size}_i) \quad m[x_{\mathbb{N}}] = \text{inl msg}_s}{\langle \text{MPtrDeref } x_p, \sigma, m \rangle \Downarrow \text{inl } "MPtrDeref \text{ lookup failed}"} \text{MPtrDerefErr2}$$

$$\frac{}{\langle \text{MConst } v_b \text{ size}_i, \sigma, m \rangle \Downarrow \text{inr } (v_b, \text{size}_i)} \text{MConst}$$

NExpr evaluation semantics

$$\frac{\sigma[x_{\mathbb{N}}] = \text{inr } (\text{DSHnatVal } v_i)}{\langle \text{NVar } x_{\mathbb{N}}, \sigma, m \rangle \Downarrow \text{inr } v_i} \text{NVarOK}$$

$$\frac{\sigma[x_{\mathbb{N}}] = \text{inl msg}_s}{\langle \text{NVar } x_{\mathbb{N}}, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NVarErr1}$$

$$\frac{\sigma[x_{\mathbb{N}}] = \text{inr } (\text{DSHCTypeVal } v_c)}{\langle \text{NVar } x_{\mathbb{N}}, \sigma, m \rangle \Downarrow \text{inl } "invalid \text{ NVar type}"} \text{NVarErr2}$$

$$\frac{\sigma[x_{\mathbb{N}}] = \text{inr } (\text{DSHPtrVal } v_{\mathbb{N}} \text{ size}_i)}{\langle \text{NVar } x_{\mathbb{N}}, \sigma, m \rangle \Downarrow \text{inl } "invalid \text{ NVar type}"} \text{NVarErr3}$$

$$\frac{}{\langle \text{NConst } v_i, \sigma, m \rangle \Downarrow \text{inr } v_i} \text{NConst}$$

$$\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inr } a_i \quad \langle b_n, \sigma, m \rangle \Downarrow \text{inr } b_i \quad b_i \neq 0}{\langle \text{NDiv } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inr } \frac{a_i}{b_i}} \text{NDivOK}$$

$$\frac{\langle b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{NDiv } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NDivErr1}$$

$$\frac{\langle b_n, \sigma, m \rangle \Downarrow \text{inr } b_i \quad b_i = 0}{\langle \text{NDiv } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl } "Division \text{ by } 0"} \text{NDivErr2}$$

$$\frac{\langle b_n, \sigma, m \rangle \Downarrow \text{inr } b_i \quad b_i \neq 0 \quad \langle a_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{NDiv } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NDivErr3}$$

$$\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inr } a_i \quad \langle b_n, \sigma, m \rangle \Downarrow \text{inr } b_i \quad b_i \neq 0}{\langle \text{NMod } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inr } (a_i \bmod b_i)} \text{NModOK}$$

$$\frac{\langle b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{NMod } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NModErr1}$$

$$\frac{\langle b_n, \sigma, m \rangle \Downarrow \text{inr } b_i \quad b_i = 0}{\langle \text{NMod } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl "Mod by 0"}} \text{NModErr2}$$

$$\frac{\langle b_n, \sigma, m \rangle \Downarrow \text{inr } b_i \quad b_i \neq 0 \quad \langle a_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{NMod } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NModErr3}$$

$$\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inr } a_i \quad \langle b_n, \sigma, m \rangle \Downarrow \text{inr } b_i}{\langle \text{NPlus } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inr } (a_i + b_i)} \text{NPlusOK}$$

$$\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{NPlus } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NPlusErr1}$$

$$\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inr } a_i \quad \langle b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{NPlus } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NPlusErr2}$$

$$\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inr } a_i \quad \langle b_n, \sigma, m \rangle \Downarrow \text{inr } b_i}{\langle \text{NMinus } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inr } (a_i - b_i)} \text{NMinusOK}$$

$$\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{NMinus } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NMinusErr1}$$

$$\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inr } a_i \quad \langle b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{NMinus } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NMinusErr2}$$

$$\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inr } a_i \quad \langle b_n, \sigma, m \rangle \Downarrow \text{inr } b_i}{\langle \text{NMult } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inr } (a_i \cdot b_i)} \text{NMultOK}$$

$$\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{NMult } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NMultErr1}$$

$$\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inr } a_i \quad \langle b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{NMult } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NMultErr2}$$

$$\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inr } a_i \quad \langle b_n, \sigma, m \rangle \Downarrow \text{inr } b_i}{\langle \text{NMin } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inr } \min(a_i, b_i)} \text{NMinOK}$$

$$\begin{array}{c}
\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{NMin } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NMinErr1} \\
\\
\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inr } a_i \quad \langle b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{NMin } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NMinErr2} \\
\\
\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inr } a_i \quad \langle b_n, \sigma, m \rangle \Downarrow \text{inr } b_i}{\langle \text{NMax } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inr max}(a_i, b_i)} \text{NMaxOK} \\
\\
\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{NMax } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NMaxErr1} \\
\\
\frac{\langle a_n, \sigma, m \rangle \Downarrow \text{inr } a_i \quad \langle b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{NMax } a_n \ b_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{NMaxErr2}
\end{array}$$

AExpr evaluation semantics

$$\begin{array}{c}
\frac{\sigma[x_{\mathbb{N}}] = \text{inr (DSHCTypeVal } v_c)}{\langle \text{AVar } x_{\mathbb{N}}, \sigma, m \rangle \Downarrow \text{inr } v_c} \text{AVarOK} \\
\\
\frac{\sigma[x_{\mathbb{N}}] = \text{inl msg}_s}{\langle \text{AVar } x_{\mathbb{N}}, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{AVarErr1} \\
\\
\frac{\sigma[x_{\mathbb{N}}] = \text{inr (DSHnatVal } v_i)}{\langle \text{AVar } x_{\mathbb{N}}, \sigma, m \rangle \Downarrow \text{inl "invalid AVar type"}} \text{AVarErr2} \\
\\
\frac{\sigma[x_{\mathbb{N}}] = \text{inr (DSHPtrVal } v_{\mathbb{N}} \text{ size}_i)}{\langle \text{AVar } x_{\mathbb{N}}, \sigma, m \rangle \Downarrow \text{inl "invalid AVar type"}} \text{AVarErr3} \\
\\
\frac{}{\langle \text{AConst } v_c, \sigma, m \rangle \Downarrow \text{inr } v_c} \text{AConst} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inr } v_c}{\langle \text{AAbs } a_a, \sigma, m \rangle \Downarrow \text{inr } |v_c|} \text{AAbsOK} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{AAbs } a_a, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{AAbsErr}
\end{array}$$

$$\begin{array}{c}
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inr } a_c \quad \langle b_a, \sigma, m \rangle \Downarrow \text{inr } b_c}{\langle \text{APlus } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inr } (a_c + b_c)} \text{APlusOK} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{APlus } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{APlusErr1} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inr } a_c \quad \langle b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{APlus } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{APlusErr2} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inr } a_c \quad \langle b_a, \sigma, m \rangle \Downarrow \text{inr } b_c}{\langle \text{AMinus } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inr } (a_c - b_c)} \text{AMinusOK} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{AMinus } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{AMinusErr1} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inr } a_c \quad \langle b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{AMinus } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{AMinusErr2} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inr } a_c \quad \langle b_a, \sigma, m \rangle \Downarrow \text{inr } b_c}{\langle \text{AMult } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inr } (a_c \cdot b_c)} \text{AMultOK} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{AMult } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{AMultErr1} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inr } a_c \quad \langle b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{AMult } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{AMultErr2} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inr } a_c \quad \langle b_a, \sigma, m \rangle \Downarrow \text{inr } b_c}{\langle \text{AMin } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inr } \min(a_c, b_c)} \text{AMinOK} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{AMin } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{AMinErr1} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inr } a_c \quad \langle b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{AMin } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{AMinErr2} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inr } a_c \quad \langle b_a, \sigma, m \rangle \Downarrow \text{inr } b_c}{\langle \text{AMax } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inr } \max(a_c, b_c)} \text{AMaxOK}
\end{array}$$

$$\begin{array}{c}
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{AMax } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{AMaxErr1} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inr } a_c \quad \langle b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{AMax } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{AMaxErr2} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inr } a_c \quad \langle b_a, \sigma, m \rangle \Downarrow \text{inr } b_c \quad a_c < b_c}{\langle \text{AZless } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inr one}} \text{AZlessOKL} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inr } a_c \quad \langle b_a, \sigma, m \rangle \Downarrow \text{inr } b_c \quad a_c \geq b_c}{\langle \text{AZless } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inr zero}} \text{AZlessOKR} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{AZless } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{AZlessErr1} \\
\\
\frac{\langle a_a, \sigma, m \rangle \Downarrow \text{inr } a_c \quad \langle b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{AZless } a_a \ b_a, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{AZlessErr2} \\
\\
\frac{\langle i_n, \sigma, m \rangle \Downarrow \text{inr } i_i \quad \langle x_m, \sigma, m \rangle \Downarrow \text{inr } (x_b, \text{size}_i) \quad i_i < \text{size}_i \quad x_b[i_i] = \text{inr } v_c}{\langle \text{ANth } x_m \ i_n, \sigma, m \rangle \Downarrow \text{inr } v_c} \text{ANthOK} \\
\\
\frac{\langle i_n, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{ANth } x_m \ i_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{ANthErr1} \\
\\
\frac{\langle i_n, \sigma, m \rangle \Downarrow \text{inr } i_i \quad \langle x_m, \sigma, m \rangle \Downarrow \text{inl msg}_s}{\langle \text{ANth } x_m \ i_n, \sigma, m \rangle \Downarrow \text{inl msg}_s} \text{ANthErr2} \\
\\
\frac{\langle i_n, \sigma, m \rangle \Downarrow \text{inr } i_i \quad \langle x_m, \sigma, m \rangle \Downarrow \text{inr } (x_b, \text{size}_i) \quad i_i \geq \text{size}_i}{\langle \text{ANth } x_m \ i_n, \sigma, m \rangle \Downarrow \text{inl "ANth index out of bounds"}} \text{ANthErr3} \\
\\
\frac{\langle i_n, \sigma, m \rangle \Downarrow \text{inr } i_i \quad \langle x_m, \sigma, m \rangle \Downarrow \text{inr } (x_b, \text{size}_i) \quad i_i < \text{size}_i \quad x_b[i_i] = \text{inl msg}_s}{\langle \text{ANth } x_m \ i_n, \sigma, m \rangle \Downarrow \text{inl "ANth not in memory"}} \text{ANthErr4}
\end{array}$$

A.3 Operators

Operators evaluate in a given state (σ, m) , and the result of the evaluation is either an error signaled via an **err** monad by returning $(\text{inl } \textit{error_message})$ or the updated memory m' returned as $(\text{inr } m')$.

We write $m[\text{addr}_{\mathbb{N}}/\text{value}_b]$ for the memory state obtained by replacing a memory block at address $\text{addr}_{\mathbb{N}}$ with the new memory block value_b . To free a memory at address $\text{addr}_{\mathbb{N}}$ and designate this address as currently unassigned (which would lead to an error trying to access it) we write $m[\text{addr}_{\mathbb{N}}/]$.

Similarly, $\text{block}_b[\text{offset}_{\mathbb{N}}/\text{value}_c]$ will result in a memory block with value at $\text{offset}_{\mathbb{N}}$ replaced with value_c . Notation $\text{block}_b[0 \dots \text{offset}_{\mathbb{N}}/\text{value}_c]$ means to initialize memory block elements with offsets in the range $[0 \dots \text{offset}_n)$ with value value_c .

We use $\text{from_nat} : \mathbb{N} \rightarrow \text{err NT.t}$ function to convert natural numbers to NT.t . Such a conversion could return an error because the set of natural numbers is infinite, while NT.t is a finite set. The conversion in the other direction: from NT.t to \mathbb{N} is implicit, and we can use just NT.t where \mathbb{N} is expected.

We use the *cons* operator $::$ to prepend σ with a new element which will have index 0. For example: $(\text{DSHnatVal } n_i) :: \sigma$.

Below, we present big-step inference rules for *DHCOL* operators. For brevity, we include only rules describing successful evaluation and omit rules describing error handling. Since error handling is always done via the **err** monad, whenever the premise of a rule contains a clause in the form $x = \text{inr } y$, it should be assumed that there is a corresponding rule for the error case $x = \text{inl } \textit{msg}_s$ with $\text{inl } \textit{error_message}$ on the right side of the \Downarrow in the conclusion.

$$\frac{}{\langle \text{DSHNop}, \sigma, m \rangle \Downarrow \text{inr } m} \text{Nop}$$

$$\frac{\begin{array}{l} \langle x_p, \sigma, m \rangle \Downarrow \text{inr } (x_{\mathbb{N}}, \text{xsize}_i) \quad \langle y_p, \sigma, m \rangle \Downarrow \text{inr } (y_{\mathbb{N}}, \text{ysize}_i) \\ m[x_{\mathbb{N}}] = \text{inr } x_b \quad m[y_{\mathbb{N}}] = \text{inr } y_b \\ \langle \text{src}_n, \sigma, m \rangle \Downarrow \text{inr } \text{src}_i \quad \langle \text{dst}_n, \sigma, m \rangle \Downarrow \text{inr } \text{dst}_i \\ \text{dst}_i < \text{ysize}_i \quad x_b[\text{src}_i] = \text{inl } v_c \end{array}}{\langle \text{DSHAssign } (x_p, \text{src}_n) (y_p, \text{dst}_n), \sigma, m \rangle \Downarrow \text{inr } m[y_{\mathbb{N}}/(y_b[\text{dst}_i/v_c])]} \text{Assign}$$

$$\frac{\begin{array}{l} \langle x_p, \sigma, m \rangle \Downarrow \text{inr } (x_{\mathbb{N}}, \text{xsize}_i) \quad \langle y_p, \sigma, m \rangle \Downarrow \text{inr } (y_{\mathbb{N}}, \text{ysize}_i) \\ m[x_{\mathbb{N}}] = \text{inr } x_b \quad m[y_{\mathbb{N}}] = \text{inr } y_b \\ n_i \leq \text{xsize}_i \quad n_i = \text{from_nat } n_{\mathbb{N}} \end{array}}{\langle \text{evalIMap } n_i x_b y_b f_a, \sigma, m \rangle \Downarrow \text{inr } y'_b \quad \langle \text{DSHIMap } n_{\mathbb{N}} x_p y_p f_a, \sigma, m \rangle \Downarrow \text{inr } m[y_{\mathbb{N}}/y'_b]} \text{IMap}$$

$$\frac{}{\langle \text{evalIMap } 0 x_b y_b f_a, \sigma, m \rangle \Downarrow \text{inr } y_b} \text{evalIMap0}$$

$$\frac{\begin{array}{l} a_c = \text{inr } x_b[n_{\mathbb{N}}] \\ \sigma' = (\text{DSHCTYPEVal } a_c) :: (\text{DSHnatVal } n_i) :: \sigma \\ \langle f_a, \sigma', m \rangle \Downarrow \text{inr } v_c \end{array}}{\langle \text{evalIMap } n_i x_b y_b f_a, \sigma, y_b[n_i/v_c] \rangle \Downarrow \text{inr } y'_b \quad \langle \text{evalIMap } (n_i + 1) x_b y_b f_a, \sigma, m \rangle \Downarrow \text{inr } y'_b} \text{evalIMapSn}$$

$$\frac{\begin{array}{l} \langle x0_p, \sigma, m \rangle \Downarrow \text{inr } (x0_{\mathbb{N}}, \text{x0size}_i) \quad m[x0_{\mathbb{N}}] = \text{inr } x0_b \\ \langle x1_p, \sigma, m \rangle \Downarrow \text{inr } (x1_{\mathbb{N}}, \text{x1size}_i) \quad m[x1_{\mathbb{N}}] = \text{inr } x1_b \\ \langle y_p, \sigma, m \rangle \Downarrow \text{inr } (y_{\mathbb{N}}, \text{ysize}_i) \quad m[y_{\mathbb{N}}] = \text{inr } y_b \\ n_i = \text{from_nat } n_{\mathbb{N}} \quad n_i \leq \text{ysize}_i \end{array}}{\langle \text{evalMap2 } n_i x0_b x1_b y_b f_a, \sigma, m \rangle \Downarrow \text{inr } y'_b \quad \langle \text{DSHMemMap2 } n_{\mathbb{N}} x0_p x1_p y_p f_a, \sigma, m \rangle \Downarrow \text{inr } m[y_{\mathbb{N}}/y'_b]} \text{Map2}$$

$$\frac{}{\langle \text{evalMap2 } 0 \ x0_b \ x1_b \ y_b \ f_a, \sigma, m \rangle \Downarrow \text{inr } y_b} \text{evalMap2O}$$

$$\begin{array}{l} a_c = \text{inr } x0_b[n_{\mathbb{N}}] \\ b_c = \text{inr } x1_b[n_{\mathbb{N}}] \\ \sigma' = (\text{DSHCTypeVal } b_c) :: (\text{DSHCTypeVal } a_c) :: \sigma \\ \langle f_a, \sigma', m \rangle \Downarrow \text{inr } v_c \\ \frac{\langle \text{evalMap2 } n_i \ x0_b \ x1_b \ y_b[n_{\mathbb{N}}/v_c] \ f_a, \sigma, m \rangle \Downarrow \text{inr } y'_b}{\langle \text{evalMap2 } (n_i + 1) \ x0_b \ x1_b \ y_b \ f_a, \sigma, m \rangle \Downarrow \text{inr } y'_b} \text{evalMap2Sn} \end{array}$$

$$\begin{array}{l} \langle x_p, \sigma, m \rangle \Downarrow \text{inr } (x_{\mathbb{N}}, \text{xsize}_i) \qquad \langle y_p, \sigma, m \rangle \Downarrow \text{inr } (y_{\mathbb{N}}, \text{ysize}_i) \\ m[x_{\mathbb{N}}] = \text{inr } x_b \qquad m[y_{\mathbb{N}}] = \text{inr } y_b \\ n_i = \text{from_nat } n_{\mathbb{N}} \qquad n_i \leq \text{ysize}_i \\ \frac{\langle \text{evalBinOp } n_i \ n_i \ x_b \ y_b \ f_a, \sigma, m \rangle \Downarrow \text{inr } y'_b}{\langle \text{DSHBinOp } n_{\mathbb{N}} \ x_p \ y_p \ f_a, \sigma, m \rangle \Downarrow \text{inr } m[y_{\mathbb{N}}/y'_b]} \text{BinOp} \end{array}$$

$$\frac{}{\langle \text{evalBinOp } 0 \ \text{off}_i \ x_b \ y_b \ f_a, \sigma, m \rangle \Downarrow \text{inr } y_b} \text{evalBinOpO}$$

$$\begin{array}{l} a_c = \text{inr } x_b[n_i] \\ b_c = \text{inr } x_b[n_i + \text{off}_i] \\ \sigma' = (\text{DSHCTypeVal } b_c) :: (\text{DSHCTypeVal } a_c) :: (\text{DSHnatVal } n_i) :: \sigma \\ \langle f_a, \sigma', m \rangle \Downarrow \text{inr } v_c \\ \frac{\langle \text{evalBinOp } n_i \ \text{off}_i \ x_b \ y_b[n_i/v_c] \ f_a, \sigma, m \rangle \Downarrow \text{inr } y'_b}{\langle \text{evalBinOp } (n_i + 1) \ \text{off}_i \ x_b \ y_b \ f_a, \sigma, m \rangle \Downarrow \text{inr } y'_b} \text{evalBinOpSn} \end{array}$$

$$\begin{array}{l} \langle x_p, \sigma, m \rangle \Downarrow \text{inr } (x_{\mathbb{N}}, \text{xsize}_i) \qquad \langle y_p, \sigma, m \rangle \Downarrow \text{inr } (y_{\mathbb{N}}, \text{ysize}_i) \\ m[x_{\mathbb{N}}] = \text{inr } x_b \qquad m[y_{\mathbb{N}}] = \text{inr } y_b \\ \langle \text{src}_n, \sigma, m \rangle \Downarrow \text{inr } \text{src}_i \qquad \langle \text{dst}_n, \sigma, m \rangle \Downarrow \text{inr } \text{dst}_i \\ \text{dst}_i < \text{ysize}_i \\ \frac{\langle \text{evalPower } n_i \ x_b \ y_b \ \text{src}_i \ \text{dst}_i \ f_a \ \sigma, m[\text{dst}_i/\text{initial}_c] \rangle \Downarrow \text{inr } y'_b}{\langle \text{DSHPower } n_n \ (x_p, \text{src}_n) \ (y_p, \text{dst}_n) \ f_a \ \text{initial}_c, \sigma, m \rangle \Downarrow \text{inr } m[y_{\mathbb{N}}/y'_b]} \text{Power} \end{array}$$

$$\frac{}{\langle \text{evalPower } 0 \ x_b \ y_b \ \text{src}_i \ \text{dst}_i \ f_a \ \sigma, m \rangle \Downarrow \text{inr } y_b} \text{evalPowerO}$$

$$a_c = x_b[\text{src}_i]$$

$$b_c = y_b[\text{dst}_i]$$

$$\sigma' = (\text{DSHCTypeVal } b_c) :: (\text{DSHCTypeVal } a_c) :: \sigma$$

$$\langle f_a, \sigma', m \rangle \Downarrow \text{inr } v_c$$

$$\frac{\langle \text{evalPower } n_i \ x_b \ y_b[\text{dst}_i/v_c] \ \text{src}_i \ \text{dst}_i \ f_a \ \sigma, m \rangle \Downarrow \text{inr } y'_b}{\langle \text{evalPower } (n_i + 1) \ x_b \ y_b \ \text{src}_i \ \text{dst}_i \ f_a \ \sigma, m \rangle \Downarrow \text{inr } y'_b} \text{evalPowerSn}$$

$$\frac{}{\langle \text{DSHLoop } 0 \ \text{body}, \sigma, m \rangle \Downarrow \text{inr } m} \text{LoopO}$$

$$n_i = \text{from_nat } n_{\mathbb{N}}$$

$$\langle \text{DSHLoop } n_{\mathbb{N}} \ \text{body}, \sigma, m \rangle \Downarrow \text{inr } m'$$

$$\frac{\langle \text{body}, ((\text{DSHnatVal } n_i) :: \sigma), m' \rangle \Downarrow \text{inr } m''}{\langle \text{DSHLoop } (n_{\mathbb{N}} + 1) \ \text{body}, \sigma, m \rangle \Downarrow \text{inr } m''} \text{LoopSn}$$

$$\frac{\langle f, \sigma, m \rangle \Downarrow \text{inr } m' \quad \langle g, \sigma, m' \rangle \Downarrow \text{inr } m''}{\langle \text{DSHSeq } f \ g, \sigma, m \rangle \Downarrow \text{inr } m''} \text{Seq}$$

$$\frac{\langle y_p, \sigma, m \rangle \Downarrow \text{inr } (y_{\mathbb{N}}, \text{ysize}_i) \quad y_b = \text{inr } m[y_{\mathbb{N}}]}{\langle \text{DSHMemInit } y_p \ v_c, \sigma, m \rangle \Downarrow \text{inr } m[y_{\mathbb{N}}/(y_b[0 \dots \text{ysize}_i/v_c])]} \text{MemInit}$$

$$\frac{\exists t_i, m[t_i] = \text{inl } \text{msg}_s \quad \langle \text{body}, ((\text{DSHPtrVal } t_i \ \text{size}_i) :: \sigma), m[t_i/\text{mem.empty}] \rangle \Downarrow \text{inr } m'[t_i/]}{\langle \text{DSHAlloc } \text{size}_i \ \text{body}, \sigma, m \rangle \Downarrow \text{inr } m'} \text{Alloc}$$

Appendix B

Examples of Generated Code

B.1 Dynamic Window Monitor in LLVM IR

HELIX-generated LLVM IR code for the dynamic window monitor is shown in Listing B.1. For legibility, we wrapped some long lines using a *backslash* symbol “\” to indicate where lines have been split. This is not a part of official IR syntax.

```
; Global variables
@D = external constant [3 x double], align 16

; Prototypes for intrinsics we use
declare float @llvm.fabs.f32(float)
declare double @llvm.fabs.f64(double)
declare float @llvm.maxnum.f32(float, float)
declare double @llvm.maxnum.f64(double, double)
declare float @minimum.f32(float, float)
declare double @llvm.minimum.f64(double, double)
declare void @llvm.memcpy.p0i8.p0i8.i32(i8*, i8*, i32, i32, i1)

; Top-level operator definition
define void @dynwin64([5 x double]* readonly align 16 nonnull %X, \
                      [1 x double]* align 16 nonnull %Y) \
```

```

{
; --- Operator: DSHAlloc 2---
b56:

    %a0 = alloca [2 x double], align 16
    br label %b55
; --- Operator: DSHAlloc 1---
; --- Operator: DSHSeq---
; --- Operator: DSHSeq---
b55:

    %a66 = alloca [1 x double], align 16
    br label %b54
; --- Operator: DSHAlloc 1---
; --- Operator: DSHSeq---
b54:

    %a70 = alloca [1 x double], align 16
    br label %MemInit_loop_entry52
; --- Operator: DSHMemInit (PVar 0) ...---
; --- Operator: DSHSeq---
MemInit_loop_entry52:

    %l111 = icmp ult i64 0, 1
    br i1 %l111, label %MemInit_loop_loop53, label %Loop_loop_entry48
MemInit_loop_loop53:
    %MemInit_init_i110 = phi i64 [0, %MemInit_loop_entry52], \
                                [%MemInit_loop_next_i113, %MemInit_init_lcont51]

    br label %MemInit_init50
MemInit_init50:

```

```

%l109 = getelementptr [1 x double], \
                                [1 x double]* %a70, \
                                i64 0, i64 %MemInit_init_i110
; void instr 64
store double 0x0, double* %l109, align 8
br label %MemInit_init_lcont51
MemInit_init_lcont51:

%MemInit_loop_next_i113 = add i64 %MemInit_init_i110, 1
%l112 = icmp ult i64 %MemInit_loop_next_i113, 1
br i1 %l112, label %MemInit_loop_loop53, label %Loop_loop_entry48
; --- Operator: DSHLoop 3 ---
Loop_loop_entry48:

%l106 = icmp ult i64 0, 3
br i1 %l106, label %Loop_loop_loop49, label %Assign31
Loop_loop_loop49:
%Loop_i71 = phi i64 [0, %Loop_loop_entry48], [%Loop_loop_next_i108, %Loop_lcont32]

br label %b47
; --- Operator: DSHAlloc 1---
; --- Operator: DSHSeq---
b47:

%a82 = alloca [1 x double], align 16
br label %Assign46
; --- Operator: DSHAssign ((PVar 7),0) ((PVar 0),0) ---
; --- Operator: DSHSeq---
Assign46:

```

```

%l103 = getelementptr [5 x double], [5 x double]* %X, i64 0, i64 0
%l105 = load double, double* %l103, align 8
%l104 = getelementptr [1 x double], [1 x double]* %a82, i64 0, i64 0
; void instr 58
store double %l105, double* %l104, align 8
br label %b45
; --- Operator: DSHAlloc 1---
b45:

%a83 = alloca [1 x double], align 16
br label %Power_entry43
; --- Operator: DSHPower (NVar 2) ((PVar 1),0) ((PVar 0),0)...---
; --- Operator: DSHSeq---
Power_entry43:

%l195 = getelementptr [1 x double], [1 x double]* %a83, i64 0, i64 0
; void instr 49
store double 0x3FF0000000000000, double* %l195, align 8
%l100 = icmp ult i64 0, %Loop_i71
br i1 %l100, label %Power_loop44, label %IMap_entry39
Power_loop44:
%Power_i94 = phi i64 [0, %Power_entry43], [%Power_next_i102, %Power_lcont41]

br label %PowerLoopBody42
PowerLoopBody42:

%l196 = getelementptr [1 x double], [1 x double]* %a82, i64 0, i64 0
%l198 = load double, double* %l196, align 8
%l197 = load double, double* %l195, align 8
%l199 = fmul double %l197, %l198
; void instr 51

```

```

store double %199, double* %195, align 8
br label %Power_lcont41
Power_lcont41:

%Power_next_i102 = add i64 %Power_i94, 1
%l101 = icmp ult i64 %Power_next_i102, %Loop_i71
br i1 %l101, label %Power_loop44, label %IMap_entry39
; --- Operator: DSHIMap 1 (PVar 0) (PVar 4) ...---
IMap_entry39:

%191 = icmp ult i64 0, 1
br i1 %191, label %IMap_loop40, label %MemMap2_entry35
IMap_loop40:
%IMap_i84 = phi i64 [0, %IMap_entry39], [%IMap_next_i93, %IMap_lcont37]

br label %IMapLoopBody38
IMapLoopBody38:

%185 = getelementptr [1 x double], [1 x double]* %a83, i64 0, i64 %IMap_i84
%187 = load double, double* %185, align 8
%188 = getelementptr [3 x double], [3 x double]* @D, i64 0, i64 %Loop_i71
%189 = load double, double* %188, align 8
%190 = fmul double %187, %189
%186 = getelementptr [1 x double], [1 x double]* %a66, i64 0, i64 %IMap_i84
; void instr 45
store double %190, double* %186, align 8
br label %IMap_lcont37
IMap_lcont37:

%IMap_next_i93 = add i64 %IMap_i84, 1
%192 = icmp ult i64 %IMap_next_i93, 1

```

```

    br i1 %192, label %IMap_loop40, label %MemMap2_entry35
; --- Operator: DSHMemMap2 1 (PVar 1) (PVar 2) (PVar 2) ...---
MemMap2_entry35:

    %179 = icmp ult i64 0, 1
    br i1 %179, label %MemMap2_loop36, label %Loop_lcont32
MemMap2_loop36:
    %MemMap2_i72 = phi i64 [0, %MemMap2_entry35], [%MemMap2_next_i81, %MemMap2_lcont33]

    br label %MemMap2LoopBody34
MemMap2LoopBody34:

    %173 = getelementptr [1 x double], [1 x double]* %a70, i64 0, i64 %MemMap2_i72
    %176 = load double, double* %173, align 8
    %174 = getelementptr [1 x double], [1 x double]* %a66, i64 0, i64 %MemMap2_i72
    %177 = load double, double* %174, align 8
    %178 = fadd double %176, %177
    %175 = getelementptr [1 x double], [1 x double]* %a66, i64 0, i64 %MemMap2_i72
; void instr 40
    store double %178, double* %175, align 8
    br label %MemMap2_lcont33
MemMap2_lcont33:

    %MemMap2_next_i81 = add i64 %MemMap2_i72, 1
    %180 = icmp ult i64 %MemMap2_next_i81, 1
    br i1 %180, label %MemMap2_loop36, label %Loop_lcont32
Loop_lcont32:

    %Loop_loop_next_i108 = add i64 %Loop_i71, 1
    %1107 = icmp ult i64 %Loop_loop_next_i108, 3
    br i1 %1107, label %Loop_loop_loop49, label %Assign31

```

```

; --- Operator: DSHAssign ((PVar 0),0) ((PVar 1),0) ---
Assign31:

%l67 = getelementptr [1 x double], [1 x double]* %a66, i64 0, i64 0
%l69 = load double, double* %l67, align 8
%l68 = getelementptr [2 x double], [2 x double]* %a0, i64 0, i64 0
; void instr 38
store double %l69, double* %l68, align 8
br label %b30

; --- Operator: DSHAlloc 1---
b30:

%a13 = alloca [1 x double], align 16
br label %b29

; --- Operator: DSHAlloc 1---
; --- Operator: DSHSeq---
b29:

%a17 = alloca [1 x double], align 16
br label %MemInit_loop_entry27

; --- Operator: DSHMemInit (PVar 0) ...---
; --- Operator: DSHSeq---
MemInit_loop_entry27:

%l63 = icmp ult i64 0, 1
br i1 %l63, label %MemInit_loop_loop28, label %Loop_loop_entry23
MemInit_loop_loop28:

%MemInit_init_i62 = phi i64 [0, %MemInit_loop_entry27], \
    [%MemInit_loop_next_i65, %MemInit_init_lcont26]

br label %MemInit_init25

```

```

MemInit_init25:

%l61 = getelementptr [1 x double], [1 x double]* %a17, i64 0, i64 %MemInit_init_i62
; void instr 31
store double 0x0, double* %l61, align 8
br label %MemInit_init_lcont26
MemInit_init_lcont26:

%MemInit_loop_next_i65 = add i64 %MemInit_init_i62, 1
%l64 = icmp ult i64 %MemInit_loop_next_i65, 1
br i1 %l64, label %MemInit_loop_loop28, label %Loop_loop_entry23
; --- Operator: DSHLoop 2 ---
Loop_loop_entry23:

%l58 = icmp ult i64 0, 2
br i1 %l58, label %Loop_loop_loop24, label %Assign6
Loop_loop_loop24:

%Loop_i18 = phi i64 [0, %Loop_loop_entry23], [%Loop_loop_next_i60, %Loop_lcont7]

br label %b22
; --- Operator: DSHAlloc 2---
; --- Operator: DSHSeq---
b22:

%a29 = alloca [2 x double], align 16
br label %Loop_loop_entry20
; --- Operator: DSHLoop 2 ---
; --- Operator: DSHSeq---
Loop_loop_entry20:

%l55 = icmp ult i64 0, 2

```



```

    br i1 %l55, label %Loop_loop_loop21, label %BinOp_entry14
Loop_loop_loop21:
    %Loop_i42 = phi i64 [0, %Loop_loop_entry20], [%Loop_loop_next_i57, %Loop_lcont16]

    br label %b19
; --- Operator: DSHalloc 1---
b19:

    %a43 = alloca [1 x double], align 16
    br label %Assign18
; --- Operator: DSHAssign ((PVar 9),?) ((PVar 0),0) ---
; --- Operator: DSHSeq---
Assign18:

    %l50 = mul i64 %Loop_i18, 1
    %l51 = add i64 1, %l50
    %l52 = mul i64 2, 1
    %l53 = mul i64 %Loop_i42, %l52
    %l54 = add i64 %l51, %l53
    %l47 = getelementptr [5 x double], [5 x double]* %X, i64 0, i64 %l54
    %l49 = load double, double* %l47, align 8
    %l48 = getelementptr [1 x double], [1 x double]* %a43, i64 0, i64 0
    ; void instr 21
    store double %l49, double* %l48, align 8
    br label %Assign17
; --- Operator: DSHAssign ((PVar 0),0) ((PVar 2),(NVar 1)) ---
Assign17:

    %l44 = getelementptr [1 x double], [1 x double]* %a43, i64 0, i64 0
    %l46 = load double, double* %l44, align 8
    %l45 = getelementptr [2 x double], [2 x double]* %a29, i64 0, i64 %Loop_i42

```

```

; void instr 19
store double %l46, double* %l45, align 8
br label %Loop_lcont16
Loop_lcont16:

%Loop_loop_next_i57 = add i64 %Loop_i42, 1
%l56 = icmp ult i64 %Loop_loop_next_i57, 2
br i1 %l56, label %Loop_loop_loop21, label %BinOp_entry14
; --- Operator: DSHBinOp 1 (PVar 0) (PVar 3) ...---
BinOp_entry14:

%l39 = icmp ult i64 0, 1
br i1 %l39, label %BinOp_loop15, label %MemMap2_entry10
BinOp_loop15:
%BinOp_i30 = phi i64 [0, %BinOp_entry14], [%BinOp_next_i41, %BinOp_lcont12]

br label %BinOpLoopBody13
BinOpLoopBody13:

%l32 = getelementptr [2 x double], [2 x double]* %a29, i64 0, i64 %BinOp_i30
%l35 = load double, double* %l32, align 8
%l31 = add i64 %BinOp_i30, 1
%l33 = getelementptr [2 x double], [2 x double]* %a29, i64 0, i64 %l31
%l36 = load double, double* %l33, align 8
%l37 = fsub double %l35, %l36
%l38 = call double @llvm.fabs.f64(double %l37)
%l34 = getelementptr [1 x double], [1 x double]* %a13, i64 0, i64 %BinOp_i30
; void instr 14
store double %l38, double* %l34, align 8
br label %BinOp_lcont12
BinOp_lcont12:

```

```

%BinOp_next_i41 = add i64 %BinOp_i30, 1
%l40 = icmp ult i64 %BinOp_next_i41, 1
br i1 %l40, label %BinOp_loop15, label %MemMap2_entry10
; --- Operator: DSHMemMap2 1 (PVar 1) (PVar 2) (PVar 2) ...---
MemMap2_entry10:

%l26 = icmp ult i64 0, 1
br i1 %l26, label %MemMap2_loop11, label %Loop_lcont7
MemMap2_loop11:
%MemMap2_i19 = phi i64 [0, %MemMap2_entry10], [%MemMap2_next_i28, %MemMap2_lcont8]

br label %MemMap2LoopBody9
MemMap2LoopBody9:

%l20 = getelementptr [1 x double], [1 x double]* %a17, i64 0, i64 %MemMap2_i19
%l23 = load double, double* %l20, align 8
%l21 = getelementptr [1 x double], [1 x double]* %a13, i64 0, i64 %MemMap2_i19
%l24 = load double, double* %l21, align 8
%l25 = call double @llvm.maxnum.f64(double %l23, double %l24)
%l22 = getelementptr [1 x double], [1 x double]* %a13, i64 0, i64 %MemMap2_i19
; void instr 9
store double %l25, double* %l22, align 8
br label %MemMap2_lcont8
MemMap2_lcont8:

%MemMap2_next_i28 = add i64 %MemMap2_i19, 1
%l27 = icmp ult i64 %MemMap2_next_i28, 1
br i1 %l27, label %MemMap2_loop11, label %Loop_lcont7
Loop_lcont7:

```

```

%Loop_loop_next_i60 = add i64 %Loop_i18, 1
%l59 = icmp ult i64 %Loop_loop_next_i60, 2
br i1 %l59, label %Loop_loop_loop24, label %Assign6
; --- Operator: DSHAssign ((PVar 0),0) ((PVar 1),1) ---
Assign6:

%l14 = getelementptr [1 x double], [1 x double]* %a13, i64 0, i64 0
%l16 = load double, double* %l14, align 8
%l15 = getelementptr [2 x double], [2 x double]* %a0, i64 0, i64 1
; void instr 7
store double %l16, double* %l15, align 8
br label %BinOp_entry4
; --- Operator: DSHBinOp 1 (PVar 0) (PVar 2) ...---
BinOp_entry4:

%l10 = icmp ult i64 0, 1
br i1 %l10, label %BinOp_loop5, label %b0
BinOp_loop5:
%BinOp_i1 = phi i64 [0, %BinOp_entry4], [%BinOp_next_i12, %BinOp_lcont2]

br label %BinOpLoopBody3
BinOpLoopBody3:

%l3 = getelementptr [2 x double], [2 x double]* %a0, i64 0, i64 %BinOp_i1
%l6 = load double, double* %l3, align 8
%l2 = add i64 %BinOp_i1, 1
%l4 = getelementptr [2 x double], [2 x double]* %a0, i64 0, i64 %l2
%l7 = load double, double* %l4, align 8
%l8 = fcmp olt double %l6, %l7
; void instr 2
; Casting bool to float

```

```

%19 = uitofp i1 %18 to double
%15 = getelementptr [1 x double], [1 x double]* %Y, i64 0, i64 %BinOp_i1
; void instr 1
store double %19, double* %15, align 8
br label %BinOp_lcont2
BinOp_lcont2:

%BinOp_next_i12 = add i64 %BinOp_i1, 1
%l11 = icmp ult i64 %BinOp_next_i12, 1
br i1 %l11, label %BinOp_loop5, label %b0
b0:

ret void
}

```

Listing B.1: Dynamic Window Monitor in LLVM IR

B.2 Dynamic Window Monitor in C

Listing B.2: Dynamic Window Monitor in C (not optimized)

```

int dwmonitor(const double *X, const double *D) {
    double q3, q4, s1, s4, s5, s6, s7, s8, w1;
    int w2;
    s5 = 0.0;
    s8 = X[0];
    s7 = 1.0;
    for (int i = 0; i <= 2; i++) {
        s4 = s7 * D[i];
        s5 = s5 + s4;
        s7 = s7 * s8;
    }
    s1 = 0.0;
    for (int i = 0; i <= 1; i++) {

```

```

    q3 = X[i + 1];
    q4 = X[3 + i];
    w1 = q3 - q4;
    s6 = (w1 >= 0) ? w1 : (-w1);
    s1 = (s1 >= s6) ? s1 : s6;
}
w2 = (s1 >= s5);
return w2;
}

```

Listing B.3: Dynamic Window Monitor in C (optimized)

```

#include <float.h>
#include <smmmintrin.h>

int dwmonitor(float *X, double *D) {
    __m128d u1, u2, u3, u4, u5, u6, u7, u8, x1, x10, x13, x14, x17, x18, x19, x2,
        x3, x4, x6, x7, x8, x9;
    int w1;
    {
        unsigned _xm = _mm_getcsr();
        _mm_setcsr((_xm & 0xffff0000) | 0x0000dfc0);
        u5 = _mm_set1_pd(0.0);
        u2 = _mm_cvtps_pd(_mm_addsub_ps(_mm_set1_ps(FLT_MIN), _mm_set1_ps(X[0])));
        u1 = _mm_set_pd(1.0, (-1.0));
        for (int i5 = 0; i5 <= 2; i5++) {
            x6 = _mm_addsub_pd(_mm_set1_pd((DBL_MIN + DBL_MIN)),
                _mm_loadup_pd(&(D[i5])));
            x1 = _mm_addsub_pd(_mm_set1_pd(0.0), u1);
            x2 = _mm_mul_pd(x1, x6);
            x3 = _mm_mul_pd(_mm_shuffle_pd(x1, x1, _MM_SHUFFLE2(0, 1)), x6);
            x4 = _mm_sub_pd(_mm_set1_pd(0.0), _mm_min_pd(x3, x2));
            u3 =
                _mm_add_pd(_mm_max_pd(_mm_shuffle_pd(x4, x4, _MM_SHUFFLE2(0, 1)), x3),
                    _mm_set1_pd(DBL_MIN));
            u5 = _mm_add_pd(u5, u3);
            x7 = _mm_addsub_pd(_mm_set1_pd(0.0), u1);
            x8 = _mm_mul_pd(x7, u2);
            x9 = _mm_mul_pd(_mm_shuffle_pd(x7, x7, _MM_SHUFFLE2(0, 1)), u2);
            x10 = _mm_sub_pd(_mm_set1_pd(0.0), _mm_min_pd(x9, x8));
            u1 = _mm_add_pd(
                _mm_max_pd(_mm_shuffle_pd(x10, x10, _MM_SHUFFLE2(0, 1)), x9),
                _mm_set1_pd(DBL_MIN));
        }
    }
}

```

```

}
u6 = _mm_set1_pd(0.0);
for (int i3 = 0; i3 <= 1; i3++) {
    u8 = _mm_cvtps_pd(
        _mm_addsub_ps(_mm_set1_ps(FLT_MIN), _mm_set1_ps(X[(i3 + 1)])));
    u7 = _mm_cvtps_pd(
        _mm_addsub_ps(_mm_set1_ps(FLT_MIN), _mm_set1_ps(X[(3 + i3)])));
    x14 = _mm_add_pd(u8, _mm_shuffle_pd(u7, u7, _MM_SHUFFLE2(0, 1)));
    x13 = _mm_shuffle_pd(x14, x14, _MM_SHUFFLE2(0, 1));
    u4 = _mm_shuffle_pd(_mm_min_pd(x14, x13), _mm_max_pd(x14, x13),
        _MM_SHUFFLE2(1, 0));
    u6 = _mm_shuffle_pd(_mm_min_pd(u6, u4), _mm_max_pd(u6, u4),
        _MM_SHUFFLE2(1, 0));
}
x17 = _mm_addsub_pd(_mm_set1_pd(0.0), u6);
x18 = _mm_addsub_pd(_mm_set1_pd(0.0), u5);
x19 = _mm_cmpge_pd(x17, _mm_shuffle_pd(x18, x18, _MM_SHUFFLE2(0, 1)));
w1 = (_mm_testc_si128(
    _mm_castpd_si128(x19),
    _mm_set_epi32(0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff)) -
    (_mm_testnzc_si128(
        _mm_castpd_si128(x19),
        _mm_set_epi32(0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff))));
asm volatile("" ::: "memory");
if (_mm_getcsr() & 0x0d) {
    _mm_setcsr(_xm);
    return -1;
}
_mm_setcsr(_xm);
}
return w1;
}

```

Bibliography

- [1] R. N. Charette, “This car runs on code,” *IEEE spectrum*, vol. 46, no. 3, p. 3, 2009. [1.1](#)
- [2] C. Ebert and C. Jones, “Embedded software: Facts, figures, and future,” *Computer*, vol. 42, p. 42–52, Apr. 2009. [1.1](#)
- [3] M. Püschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, “SPIRAL: Code generation for DSP transforms,” *Proceedings of the IEEE*, vol. 93, pp. 232–275, Feb 2005. [1.2](#)
- [4] F. Franchetti, T.-M. Low, T. Popovici, R. Veras, D. G. Spampinato, J. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura, “SPIRAL: Extreme performance portability,” *Proceedings of the IEEE, special issue on “From High Level Specification to High Performance Code”*, vol. 106, no. 11, 2018. [1.2](#)
- [5] F. Franchetti, Y. Voronenko, and M. Püschel, “Formal loop merging for signal transforms,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’05, (New York, NY, USA), pp. 315–326, ACM, 2005. [1.2](#)
- [6] F. Franchetti, T. M. Low, S. Mitsch, J. P. Mendoza, L. Gui, A. Phaosawasdi, D. Padua, S. Kar, J. M. F. Moura, M. Franusich, J. Johnson, A. Platzer, and

- M. M. Veloso, “High-assurance SPIRAL: End-to-end guarantees for robot and car control,” *IEEE Control Systems*, vol. 37, pp. 82–103, April 2017. [1.3](#), [1.4](#), [1.4](#), [1.4](#)
- [7] T.-M. Low and F. Franchetti, “High assurance code generation for cyber-physical systems,” in *IEEE International Symposium on High Assurance Systems Engineering (HASE)*, 2017. [1.3](#)
- [8] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.11.0*, 2020. [1.3](#)
- [9] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel, “Operator language: A program generation framework for fast kernels,” in *IFIP Working Conference on Domain Specific Languages (DSL WC)*, vol. 5658 of *Lecture Notes in Computer Science*, pp. 385–410, Springer, 2009. [1.3](#)
- [10] D. Fox, W. Burgard, and S. Thrun, “The dynamic window approach to collision avoidance,” *IEEE Robotics & Automation Magazine*, vol. 4, no. 1, pp. 23–33, 1997. [1.4](#)
- [11] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völpl, and A. Platzer, “Keymaera x: An axiomatic tactical theorem prover for hybrid systems,” in *International Conference on Automated Deduction*, pp. 527–538, Springer, 2015. [1.4](#)
- [12] A. W. Appel, L. Beringer, A. Chlipala, B. C. Pierce, Z. Shao, S. Weirich, and S. Zdancewic, “Position paper: the science of deep specification,” *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 375, no. 2104, p. 20160331, 2017. [1.4](#)
- [13] B. Russel, *Principia mathematica*. University Press, 1950. [2.2](#)

- [14] J. J. Kohfeld, “A constructive proof of the fundamental theorem of algebra,” Master’s thesis, Oregon State College, 1959. [2.2](#)
- [15] H. Geuvers, F. Wiedijk, and J. Zwanenburg, “A constructive proof of the fundamental theorem of algebra without using the rationals,” in *International Workshop on Types for Proofs and Programs*, pp. 96–111, Springer, 2000. [2.2](#)
- [16] L. Cruz-Filipe, “A constructive formalization of the fundamental theorem of calculus,” in *International Workshop on Types for Proofs and Programs*, pp. 108–126, Springer, 2002. [2.2](#), [2.3](#)
- [17] P. Martin-Löf, “Notes on constructive mathematics,” *Almqvist & Amp*, 1970. [2.2](#)
- [18] T. Coquand and G. Huet, *The calculus of constructions*. PhD thesis, INRIA, 1986. [2.2](#)
- [19] T. C. development team, *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0. [2.2](#), [3](#)
- [20] B. C. Pierce, *Advanced topics in types and programming languages*. MIT press, 2005. [2.2](#)
- [21] M. H. B. Sørensen and P. Urzyczyn, “Lectures on the Curry-Howard Isomorphism,” 1998. [2.2](#)
- [22] M. Sozeau, Y. Forster, S. Boulier, T. Winterhalter, and N. Tabareau, “Coq Coq Codet! Towards a Verified Toolchain for Coq in MetaCoq (slides),” 2019. [2.3](#)
- [23] M. Oostdijk and H. Geuvers, “Proof by computation in the Coq system,” *Theoretical Computer Science*, vol. 272, no. 1, pp. 293 – 314, 2002. Theories of Types and Proofs 1997. [2.3](#)

- [24] X. Leroy, “Formal verification of a realistic compiler,” *Commun. ACM*, vol. 52, pp. 107–115, July 2009. [2.3](#), [2.5](#), [3.3.1](#)
- [25] A. Chlipala, “The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier,” *SIGPLAN Not.*, vol. 48, p. 391–402, Sept. 2013. [2.3](#)
- [26] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O’Connor, S. O. Biha, *et al.*, “A machine-checked proof of the odd order theorem,” in *International Conference on Interactive Theorem Proving*, pp. 163–179, Springer, 2013. [2.3](#)
- [27] G. Gonthier, “Formal proof—the four-color theorem,” *Notices of the AMS*, vol. 55, no. 11, pp. 1382–1393, 2008. [2.3](#)
- [28] A. Bauer, J. Gross, P. L. Lumsdaine, M. Shulman, M. Sozeau, and B. Spitters, “The hott library: a formalization of homotopy type theory in coq,” in *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pp. 164–172, 2017. [2.3](#)
- [29] G. Pîrlea and I. Sergey, “Mechanising blockchain consensus,” in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, (New York, NY, USA), p. 78–90, Association for Computing Machinery, 2018. [2.3](#)
- [30] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, “Formal certification of elgamal encryption,” in *International Workshop on Formal Aspects in Security and Trust*, pp. 1–19, Springer, 2008. [2.3](#)

- [31] G. Barthe, D. Hedin, S. Z. Béguelin, B. Grégoire, and S. Heraud, “A machine-checked formalization of sigma-protocols,” in *2010 23rd IEEE Computer Security Foundations Symposium*, pp. 246–260, IEEE, 2010. [2.3](#)
- [32] A. W. Appel, “Verification of a cryptographic primitive: Sha-256,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 2, pp. 1–31, 2015. [2.3](#)
- [33] G. Winskel, *The formal semantics of programming languages: an introduction*. MIT press, 1993. [2.4](#)
- [34] A. Koprowski and H. Binsztok, “Trx: A formally verified parser interpreter,” in *European Symposium on Programming*, pp. 345–365, Springer, 2010. [2.4](#)
- [35] J.-H. Jourdan, F. Pottier, and X. Leroy, “Validating lr (1) parsers,” in *European Symposium on Programming*, pp. 397–416, Springer, 2012. [2.4](#)
- [36] S. Lasser, C. Casinghino, K. Fisher, and C. Roux, “A verified ll (1) parser generator,” in *10th International Conference on Interactive Theorem Proving (ITP 2019)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019. [2.4](#)
- [37] L.-y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic, “Interaction trees: representing recursive and impure programs in Coq,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, 2019. [2.4](#), [4.7](#)
- [38] Y. Zakowski, C. Beck, V. Zaliva, I. Yoon, I. Zaichuk, and S. Zdancewic, “Modular, compositional, and executable formal semantics for LLVM IR,” 2020. Submitted to PLDI 2020. [2.4](#)

- [39] M. Sozeau, A. Anand, S. Boulrier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau, and T. Winterhalter, “The metacoq project,” *Journal of Automated Reasoning*, pp. 1–53, 2020. [3](#), [4.4.1](#)
- [40] B. Spitters and E. Van der Weegen, “Type classes for mathematics in type theory,” *Mathematical Structures in Computer Science*, vol. 21, no. 4, pp. 795–825, 2011. [3.1.1](#)
- [41] F. Blanqui and A. Koprowski, “CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates,” *Mathematical Structures in Computer Science*, vol. 21, no. 4, pp. 827–859, 2011. [3.1.3](#)
- [42] Y. N. Moschovakis, “A mathematical modeling of pure, recursive algorithms,” in *International Symposium on Logical Foundations of Computer Science*, pp. 208–229, Springer, 1989. [3.1.3](#)
- [43] S. Wolfram *et al.*, *The MATHEMATICA® book, version 4*. Cambridge university press, 1999. [3.1.3](#)
- [44] G. Malecha *et al.*, “ExtLib Coq library.” <https://github.com/coq-ext-lib/coq-ext-lib>, 2012. Accessed: 2020-08-18. [3.2.1](#)
- [45] A. Chlipala, *Formal reasoning about programs*. 2017. [3.3](#)
- [46] P. Castéran and M. Sozeau, “A gentle introduction to type classes and relations in Coq,” tech. rep., Technical Report hal-00702455, version 1, 2012. [3.3](#)
- [47] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart, “The CompCert Memory Model, Version 2,” Research Report RR-7987, INRIA, June 2012. [3.3.1](#)

- [48] S. Boldo and G. Melquiond, “Flocq: A unified library for proving floating-point algorithms in Coq,” in *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pp. 243–252, IEEE, 2011. [3.5](#)
- [49] M. Sozeau, “A new look at generalized rewriting in type theory,” *Journal of Formalized Reasoning*, pp. 1–12, 2010. [4.1](#), [4.1.2](#)
- [50] S. Dolan, L. White, and A. Madhavapeddy, “Multicore ocaml,” in *OCaml Workshop*, vol. 2, 2014. [5.1](#)
- [51] N. J. Higham, *Accuracy and stability of numerical algorithms*, vol. 80. Siam, 2002. [5.3.2](#)
- [52] G. Malecha, “Speeding up proofs with computational reflection.” <https://gmalecha.github.io/reflections/2017/speeding-up-proofs-with-computational-reflection>, 2017. Accessed: 2020-09-01. [5.3.3](#)
- [53] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen, “An overview of elan,” *Electronic Notes in Theoretical Computer Science*, vol. 15, pp. 55–70, 1998. [5.4](#)
- [54] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, “Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code,” *SIGPLAN Not.*, vol. 50, p. 205–217, Aug. 2015. [5.4](#)
- [55] B. Hagedorn, J. Lenfers, T. Kundahler, X. Qin, S. Gorlatch, and M. Steuwer, “Achieving high-performance the functional way: A functional

- pearl on expressing high-performance optimizations as rewrite strategies,” *Proc. ACM Program. Lang.*, vol. 4, Aug. 2020. [5.4](#)
- [56] P. J. Brinich, “Formal verification of spiral generated code,” Master’s thesis, Drexel University, 2020. [5.4](#)
- [57] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver, “CertiCoq: A verified compiler for Coq,” in *The Third International Workshop on Coq for Programming Languages (CoqPL)*, 2017. [5.4](#)
- [58] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, “Cakeml: a verified implementation of ml,” *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 179–191, 2014. [5.4](#)
- [59] S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan, “Functional big-step semantics,” in *European Symposium on Programming*, pp. 589–615, Springer, 2016. [5.4](#)
- [60] J. C. Reynolds, “Definitional interpreters for higher-order programming languages,” in *Proceedings of the ACM Annual Conference - Volume 2*, ACM ’72, (New York, NY, USA), p. 717–740, Association for Computing Machinery, 1972. [5.4](#)
- [61] A. Carnegie and G. Hutner, *The Autobiography of Andrew Carnegie and the Gospel of Wealth*. Signet Classics, Signet Classics, 2006.
- [62] V. Zaliva and F. Franchetti, “HELIX: A case study of a formal verification of high performance program generation,” in *Proceedings of the 7th ACM SIG-*

- PLAN International Workshop on Functional High-Performance Computing*, FHPC 2018, (New York, NY, USA), pp. 1–9, ACM, 2018.
- [63] A. Anand, S. Boulier, C. Cohen, M. Sozeau, and N. Tabareau, “Towards certified meta-programming with typed Template-Coq,” in *ITP 2018-9th Conference on Interactive Theorem Proving*, 2018.
 - [64] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, “Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code,” in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, (New York, NY, USA), p. 205–217, Association for Computing Machinery, 2015.
 - [65] A. Chlipala, “The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier,” in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP ’13, (New York, NY, USA), p. 391–402, Association for Computing Machinery, 2013.
 - [66] V. Zaliva, I. Zaichuk, and F. Franchetti, “Verified translation between purely functional and imperative domain specific languages in HELIX,” in *Proceedings of the 12th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2020.
 - [67] V. Zaliva and M. Sozeau, “Reification of shallow-embedded DSLs in Coq with automated verification,” in *International Workshop on Coq for Programming Languages (CoqPL)*, 2019.
 - [68] V. Zaliva and F. Franchetti, “Formal verification of HCOL rewriting,” 2015.