HELIX: Verified compilation of cyber-physical control systems to LLVM IR

VADIM ZALIVA
University of Cambridge, UK
(email: vz231@cl.cam.ac.uk*)*

YANNICK ZAKOWSKI
INRIA, France
(email: yannick.zakowski@inria.fr*)*

ILIA ZAICHUK
Taras Shevchenko National University, Ukraine and Digamma.ai, USA
(email: zoickx@ztld.org*)*

VALERII HUHNIN
Université Paris Cité, France and Digamma.ai, USA
(email: valerii.huhnin@etu.u-paris.fr*)*

CALVIN BECK
University of Pennsylvania, USA
(email: hobbes@seas.upenn.edu*)*

IRENE YOON
University of Pennsylvania, USA
(email: inbox@ireneyoon.com*)*

STEVE ZDANCEWIC
University of Pennsylvania, USA
(email: stevez@seas.upenn.edu*)*

Abstract

This paper presents the design of HELIX, an end-to-end verified code generation system with a focus on the intersection of high-performance and high-assurance numerical computing. The code generation can be fine-tuned to generate efficient code for a broad set of computer architectures while providing formal guarantees of the correctness of such generated code.

Using a real-life example of a cyber-physical robot system, this paper demonstrates how, by using HELIX, one can start from a high-level mathematical formulation of the problem, apply a series of algebraic transformations that target intermediate languages, and generate an efficient imperative implementation. This is done while formally verifying semantic preservation from the original formulation down to LLVM IR.

The method we used for high-performance code compilation is the algebraic transformation of vector and matrix computations into a dataflow optimised for parallel or vectorised processing on target hardware. The abstraction used to formalise and verify this technique is an operator language and accompanying 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's verification infrastructure comprises multiple intermediate languages and verification approaches, all implemented in the Coq proof assistant. In particular, it uses verified term rewriting,

translation validation, metaprogramming, verified compilation, layered monadic interpreters; it also supports application-specific uses of (verified) numerical analysis as we demonstrate via the running example.

Contents

1 Introduction

1.1 Motivation and background

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 [21]. According to the same source, the premium class automobile contains more than 100 million lines of code. Parts of this code control 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. As a consequence, there are significant ongoing engineering efforts in developing reliable embedded software at such a scale [28].

Beyond safety and reliability, there are other constraints at play in such embedded systems. Most vehicular embedded software is run on Electronic Control Units (ECUs)—these are simple processors that control various aspects of car functionality. Lower-end models utilize between 30-50 ECUs, while higher-end luxury cars might use many more. These ECUs exhibit different Instruction Set Architectures (ISAs), memory architectures, clock speeds, and data bus widths. This makes the problem of developing software for them formidable: not only does one have to write and maintain potentially 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 that controls airbag deployment must react within 15 to 40 milliseconds [21].

High-performance numerical computing is an essential part of cyber-physical systems. In the course of their operation, they employ a number of numerical computations such as PID controllers [41], DFT [55], Kalman filter [42], in real-time, oftentimes on performance-constrained hardware. Hence, there is a need for high-performance implementation of such algorithms for diverse underlying hardware. The traditional approach is to implement such algorithms in a high-level language like C and use the C compiler to generate optimized code for target platforms. An alternative approach is code synthesis, where from a high-level numerical algorithm description a code is generated that is optimized for the target hardware. It has been demonstrated [33–35, 62] that such early optimization could yield better performance results than compiler optimizations such as loop unrolling, algebraic simplification, vectorization, etc. used in the traditional approach.

Another key requirement for such software is correctness. Instead of proving the correctness of one imperative implementation of the algorithm, we are now dealing with multiple versions of the same algorithm generated for different hardware architectures. During code synthesis, the target platform capabilities (parallelism, vectorization, the timing of different operations) may inform how mathematical computations may be reshaped to best map to target hardware. The correctness of the required transformations is based on algebraic reasoning and thus require different proof techniques than those used for standard compiler optimization correctness. These techniques need to combine algebraic reasoning with numerical stability, error bounds, etc. Ultimately, the correctness proofs need to extend to synthesized code compiled to machine assembly (via a low-level representation such as LLVM IR).

Therefore problem we addressed lies in this intersection of high-performance and high-assurance numerical computing. Our goal was to build a system that enables the generation of high-performance code which is proven to be correct for a class of numeric algorithms, useful for practical applications, such as code generation for the ECU vehicular control domain described above.

To that end, this paper presents HELIX, a framework for generating high-assurance numerical software, implemented and fully verified in the Coq interactive theorem prover.

This style of formal verification, in which a program is proved meet a desired specification, is a promising way to enable the construction of high-assurance software. In this approach, a HELIX source program's behaviors are described using an *interactive theorem prover* (in the case of HELIX, Coq) and then compiler optimization and code transformation passes are proved (with a machine-checked proof) to be correct, that is, the resulting low-level code's behavior is shown to *refine* that of the source program.

One challenge in building such a framework is that the design space is quite large: there are many choices and tradeoffs to be made about how to represent a program, how to define programming language semantics, how to structure the proofs of correctness, etc. The HELIX compiler pipeline uses a novel combination of *translation validation* (in which the correctness of an optimization is proved correct for a specific input program of interest) and *verified compilation* (in which a compiler transformation itself is proved to work correctly for *all* inputs) to produce an end-to-end proof of correctness.

Another challenge in this context is correctly optimizing numerical code. For this, HELIX uses a sequence of verified program transformations. The order in which these optimizations are applied is determined by an (untrusted) oracle—the previously existing SPIRAL framework [65]—guides the optimizer. These transformations exploit a novel sparse-vector representation amenable for algebraic reasoning. HELIX also facilitates the use of application-specific analyses that can be used to justify the numerical precision properties that relate the "real-valued" computations of the source algorithm to the "floating-point" computations of the target.

HELIX paves a novel path through this design space , and the key contributions of this paper are:

(1) We demonstrate how to build and verify in Coq a compilation pipeline for SPIRAL-like mathematical expressions down to LLVM IR. This pipeline involves a variety of intermediate representations, crossing from shallow, functional embedding down to a deeply embedded representation of imperative LLVM IR code. In doing so, we combine vastly different approaches to compiler verification: oracle-based metaprogramming paired with rewriting-based translation validation, traditional semantic-preserving results between big-step interpreters, and bisimulation-based results between monadic embedding of languages.
(2) We structure HELIX to facilitate the compilation to high-performance and numerically sound low-level code. In particular, we develop a methodology to algebraically reason about partial computations using sparse vectors, which would enable downstream parallelization and vectorization. Furthermore, we demonstrate how to apply domain-specific numerical analyses that provably characterize the precision of floating-point computation relative to its source semantics with respect to real values.
(3) The final step of the compilation chain targets LLVM IR, the formalization of which is given by the preexisting Vellvm project [89]. This part of our development constitutes the largest stress test to date for both Vellvm's metatheory, as well as the underlying semantic foundations it relies on, i.e., Interaction Trees.

All our contributions are formalized and proven in Coq. The development is openly available on GitHub[1].

1.2 HELIX 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 [65]. 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

[1] https://github.com/vzaliva/helix

platforms ranging from mobile devices and multicore (desktop and server) processors to high-performance and supercomputing systems [33].

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 [32, 54], is to complement SPIRAL-generated high-performance code with formally verified correctness guarantees.

At its core, HELIX constitutes *a verified compilation chain*, formalized in the Coq proof assistant, establishing a theorem of semantic preservation between high-level specifications of mathematical algorithms in a format similar to the one taken by SPIRAL as input and LLVM-compliant code. In order to achieve this result, HELIX introduces a series of intermediate languages (as shown at Figure 1), lowering down progressively the original expression. Each translation step corresponds to the concretization of a new level of abstraction:

(1) Mathematical formula
(2) Dataflow (*HCOL*[2])
(3) Dataflow with implicit loops (Σ-*HCOL*)
(4) Dataflow with implicit loops, using memory blocks (*MSHCOL*)
(5) Imperative program (*RHCOL*)
(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 for the given problem toward 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, performed in algebraic form. For example, some computations on matrices could be re-arranged into blocks fitting the target CPU SIMD instruction size.

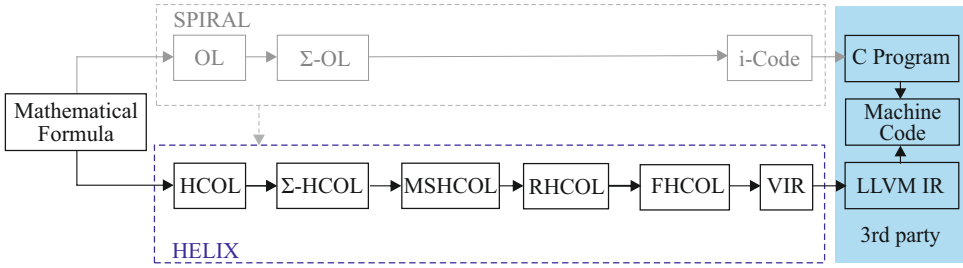

Fig. 1. HELIX transformation stages

Presently, HELIX works together with SPIRAL, which it uses as an Oracle. SPIRAL, using its extensive library of (unverified) algorithms and heuristics, decides how the original expression will be compiled into optimal code. HELIX mirrors corresponding SPIRAL transformations, adding formal verification assurances (translation validation approach [64]). The HELIX input language, called *HCOL*, is designed to be a formalization of SPIRAL's input language (*OL*), and the input program can be easily syntactically translated between the two.

[2] *HCOL* stands for Hybrid Control Operator Language.

The *HCOL* 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 [36]. 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.

- The projects have different primary goals. SPIRAL's main objective is high-performance code generation, while HELIX focuses on high assurance.
- 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 language 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. We feel that LLVM IR[3] is a better intermediate language for machine code generation. 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. Instead of generating raw IR code, HELIX creates an AST in VIR, a Vellvm-supported subset of IR. It is used in verification and subsequently to pretty-print IR source code, which could be compiled with the LLVM IR compiler.
- SPIRAL is implemented in a heavily modified [80] 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 [31]. 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

[3]The "IR" stands for *intermediate representation.*

will find the SPIRAL literature more tractable. With this paper, we aim to bridge this gap.

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. HELIX is implemented in Coq with all its languages embedded in Gallina. All translation steps are also implemented in Coq or Template-Coq [77].

A quick summary of the languages shown in Figure 1 is presented in Table 1. We chose to embed all HELIX languages in Coq Proof assistant [27], which allowed us to reason about them in Coq's Calculus of Inductive Constructions using its *Ltac* tactics language.

Name	Scalars	Vectors	Embedding	Paradigm	Error Handling
HCOL	$\mathcal{R}$	dense vector	shallow	declarative	no
Σ-*HCOL*	$\mathcal{R}_\theta$	sparse vector	mixed	functional	no
MSHCOL	$\mathcal{R}$	memory blocks	mixed	functional	yes
RHCOL	$\mathbb{R}$	env. + memory	deep	imperative	yes
FHCOL	IEEE double	env. + memory	deep	imperative	yes
LLVM IR	IEEE double	env. + memory	deep	imperative	yes

Table 1. Summary of HELIX languages

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 by variables in LLVM IR. Although all languages are embedded in Gallina, different types of embedding are used, as shown in Table 1. Also, as we proceed down the translation chain, we transition from purely functional to imperative languages. Finally, Coq's dependent type system enforces the type correctness of *HCOL*, specifically in terms of the dimensionality of input and output vectors. As a result, an *HCOL* program that successfully typechecks cannot trigger any runtime errors. However, once such a program is translated into an imperative form, error handling is introduced. The details of all these implementation aspects will be discussed for each HELIX language in detail in sections 2 to 5.

Aside from the 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 as formalizations 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 decision to use LLVM IR instead of C 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 the `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, *FHCOL*, is indeed imperative but at a higher level of abstraction than `i-Code`, more specialized, and better suited for formal reasoning and translation to IR. For example, it uses de Bruijn indices for variables, has logically scoped memory allocation, and employs a memory model similar [4] to the one used in *VIR*.

- The split between *RHCOL* and *FHCOL* is motivated by our desire to clearly define the boundary between abstract numeric types (*RHCOL*) and concrete machine types (*FHCOL*). To simplify the numeric analysis, we want these languages to be very similar, representing the same sequence of computation steps but for different types.
- The *MSHCOL* language was added for pure convenience to bridge the gap between functional Σ-*HCOL* and the imperative *RHCOL*. 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 the memory block abstraction first, while keeping the language functional.

All languages involved are embedded in Coq, but through different kinds of embedding. We chose a *shallow embedding* for *HCOL* as it reflects its algebraic nature most naturally. 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 (see Section 3.1.6) 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. *MSHCOL* representation is similar to that of Σ-*HCOL*. To express additional logical properties of operators for all these languages, we used *typeclasses*. Finally, for *RHCOL*, we switched to *deep embedding*. This allowed us to clearly separate the language's abstract syntax from its semantics and, in particular, to assign more than one semantics to the language .

The LLVM IR language formalization which we use (provided by Vellvm project [89]) is using the deep embedded representation of IR syntax. In the following, we refer to the project itself as Vellvm, and to the Coq formalization of LLVM IR that it provides as VIR, for Verified IR.

The sequence of verification steps in HELIX is shown in Figure 2 and briefly described below and again in more detail in subsequent sections.

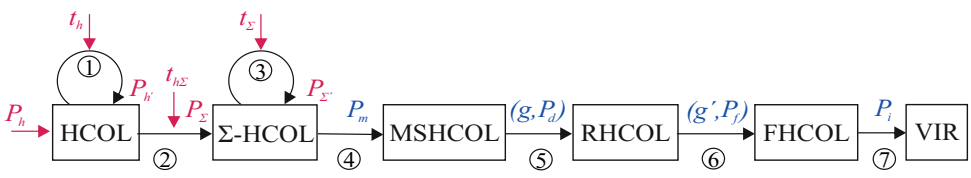

Fig. 2. HELIX chain of verification

[4] Both *FHCOL* and *VIR* use the Compcert-style memory model with integer-indexed memory blocks that contain integer-indexed memory cells. The only difference is that *FHCOL* memory cells are floats, while VIR's are bytes

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 *MSHCOL*
P_d - program in *RHCOL*
g - list of global variables upon which P_d depends along with their *RHCOL* types
P_f - program in *FHCOL*
g' - list of global variables upon which P_f depends along with their *FHCOL* types
P_i - AST of the program in LLVM IR

The numbered arrows in Figure 2 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 2.2 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 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 3.2 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 3.3 for details.

④ The *MSHCOL* 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 4.2 for details.

⑤ The *RHCOL* 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 5.5 for details.

562 ⑥ The *RHCOL* to *FHCOL* translation is implemented in Gallina. It translates program P_d
563 and a list of global variables g to the corresponding P_f and g'. This step is not
564 validated. See Section 5.7 for discussion of our reasoning.
565 ⑦ The final step of translation from *FHCOL* to LLVM IR is performed using a certified
566 compiler that we wrote in Gallina. We have proven this compiler to be correct (with
567 caveats). See Section 6.1 for details.
568

569 Thus, given an original *HCOL* expression, a SPIRAL trace file containing the transformation
570 steps, and the intermediate SPIRAL code synthesis results (all shown in red in Figure 2),
571 HELIX will either fail or provide the following high-level results:

572 (1) LLVM IR version of the *HCOL* program, which includes SPIRAL-guided code
573 optimizations.
574 (2) Correctness guarantee that the IR program on IEEE floating-point inputs that do not
575 contain NaN values will generate the same results as the *HCOL* program on real
576 numbers up to guarantees provided by the numeric analysis step.
577

578 If HELIX fails, it could be that SPIRAL-generated steps and intermediate results are
579 either inconsistent, use unproved rules, or that the HELIX LLVM compiler is missing the
580 functionality required to compile the given expression. The first indicates a bug in the SPIRAL
581 system, which needs to be fixed. The latter two reasons point to HELIX implementation
582 shortcomings that could be cured by extending HELIX with additional rules or the compiler
583 capabilities using the existing framework.
584 We discuss how these results are achieved in subsequent sections.
585

586
587

588 *1.3 Motivating example*

589 Let us consider an application of HELIX to a real-life situation of high-assurance vehicle
590 control [32] using a dynamic window vehicle control approach [30], as shown in Figure 3.
591 Given a physical model of vehicle dynamics, we generate a code to check a safety constraint.
592 In this example, we will be checking whether it's safe to proceed given the vehicle's position,
593 speed, and acceleration, with the location of an obstacle. The function will be invoked by
594 the vehicle controller, and if it returns "false," the vehicle will enter "fail-safe" mode, which
595 would trigger emergency braking.
596 We will consider a ground robot driving on a flat, even surface. The robot is equipped with
597 a distance measuring sensor (e.g. Lidar) which can detect and measure distance to obstacles.
598 The obstacle sensor is sampled periodically, for example every 20ms and based on these
599 measurements, the decisions on control outputs, such as steering, acceleration, and braking,
600 are made. Once such decisions are applied to actuators controlling the robot, no further
601 control changes can be made until the next sampling cycle. The obstacle does not have to stay
602 static and can move.
603 This model allows the development of a control system which will help a robot avoid
604 obstacles by stopping or steering around them. However, in this example, we will be
605 considering only a *safety monitor* part of the system which ensures the *passive safety* property.
606 Informally, that means there will be no collisions while the robot is driving, and collisions
607 may occur only if the obstacle runs into the robot.
608 Without discussing detailed dynamic model of the system for which we refer interested
609 readers to [32], we present the final formula for a dynamic window safety monitor in
610 Equation (1). The soundness of the monitor formula from the point of view of cyber-physical
611
612

control systems was proven in KeYmaera X [37].

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

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.

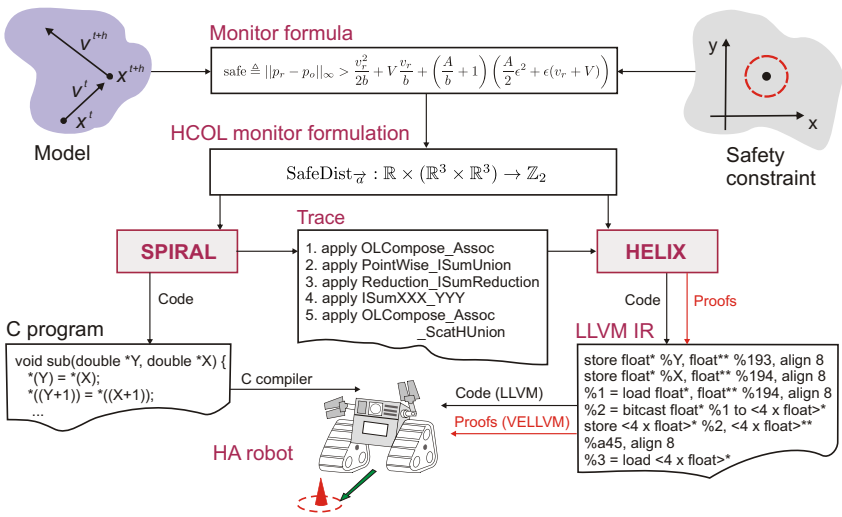

Fig. 3. HELIX application to high-assurance vehicle control

We will use HELIX to build such an implementation, while applying and certifying optimization steps produced by SPIRAL. Both HELIX and SPIRAL will start with the same input formula. Unlike SPIRAL, HELIX 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 39). This compilation will also be proven correct, and the generated code will be guaranteed to correctly correspond to the computations described by the input expression.

This final correspondence cannot be exact, due to the bounds of numerical accuracy arising from the use of floating-point calculations instead of the real precision of the original formula. Instead, HELIX provides the means for proving arbitrarily complex, abstract properties of the compilation, without focusing of numerical precision alone. In the case of *DynWin*, we take the high-level statement of "robot safety", and, relying on Equation 1 being a correct mathematical model of it, prove that, when operating on the algorithm compiled through HELIX, the robot will in fact remain "safe" at all times.

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 [4].

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.

2 *HCOL*

2.1 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*, shallow 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 defined 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 2.1.1 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 *HCOL* 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.

2.1.1 *Carrier type*

This is an abstract representation of a numeric type, expressed in terms of its operations and 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 $\mathcal{R}$. To make it abstract we define it in a type class `Carrierefs` which also postulates the existence of the following for this type (via corresponding typeclass instances):

- Equality: `Equiv` (see discussion below in Section 2.1.2.).
- Comparisons: `Lt`, `Le`.
- Decidability of the equality and comparisons: `Decision (x=y)`, `Decision (x<y)`.
- Constants: `Zero`, `One`.
- Operators: `Plus`, `Mult`, `Negate`, `Abs`.

Additional abstract algebra properties of $\mathcal{R}$, expressed using the corresponding typeclass instances from the *MathClasses* library [78] are defined in a separate typeclass `CarrierProperties`:

- Setoid Equality: `Setoid` (see discussion below in Section 2.1.2.).
- Abstract algebra structures: `Ring`.
- Ordering: `TotalOrder`, `StrictSetoidOrder`, `FullPseudoOrder`.
- Inequality of constants: `Zero ≠ One`.

The reason for splitting $\mathcal{R}$ properties into two typeclasses is to minimize the burden of typeclass resolution. The operators definitions require only basic properties from `CarrierDefs`, whereas proofs of *HCOL* program transformations add additional typeclass constraint `CarrierProperties` as a pre-condition.

In other words, 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`.

2.1.2 *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: `∀ (n:ℕ), Setoid (vector 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 paper.

2.1.3 `HPointwise` *operator*

Here is an example of an *HCOL* operator, `HPointwise`. The full list of *HCOL* operators is shown in Appendix A.2.

`HPointwise` $(n\colon \mathbb{N})\ (f\colon \mathbb{I}_n \to \mathcal{R} \to \mathcal{R})\colon \mathcal{R}^n \to \mathcal{R}^n$. This operator applies a function f to each element of the input vector, as shown in Figure 4. The function f takes two arguments: the element's index and its value. The output is the vector of the same length as the input vector.

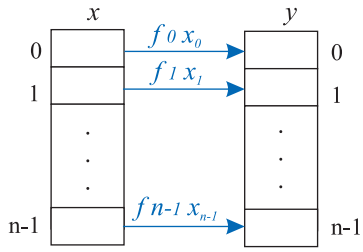

Fig. 4. `HPointwise` operator

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 1. `HPointwise` operator definition

The operator is implemented using the `Vbuild` function from the *CoLoR* librar [10]. 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.

2.1.4 Running example

An *HCOL* formulation of the dynamic window monitor expression (13) introduced in Section 1.3 is shown in Listing 2.

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

Listing 2. Dynamic Window Monitor in *HCOL*

We will use it as our running example, showing how it is translated at each step of the HELIX compilation and verification pipeline.

2.2 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* [76] together with *HCOL* operator equational theory.

The result of the *HCOL* rewriting steps of the expression from Listing 2 is shown in Listing 3

```
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 3. Dynamic Window Monitor in *HCOL* after rewriting

It corresponds to the dataflow graph shown in Figure 5.

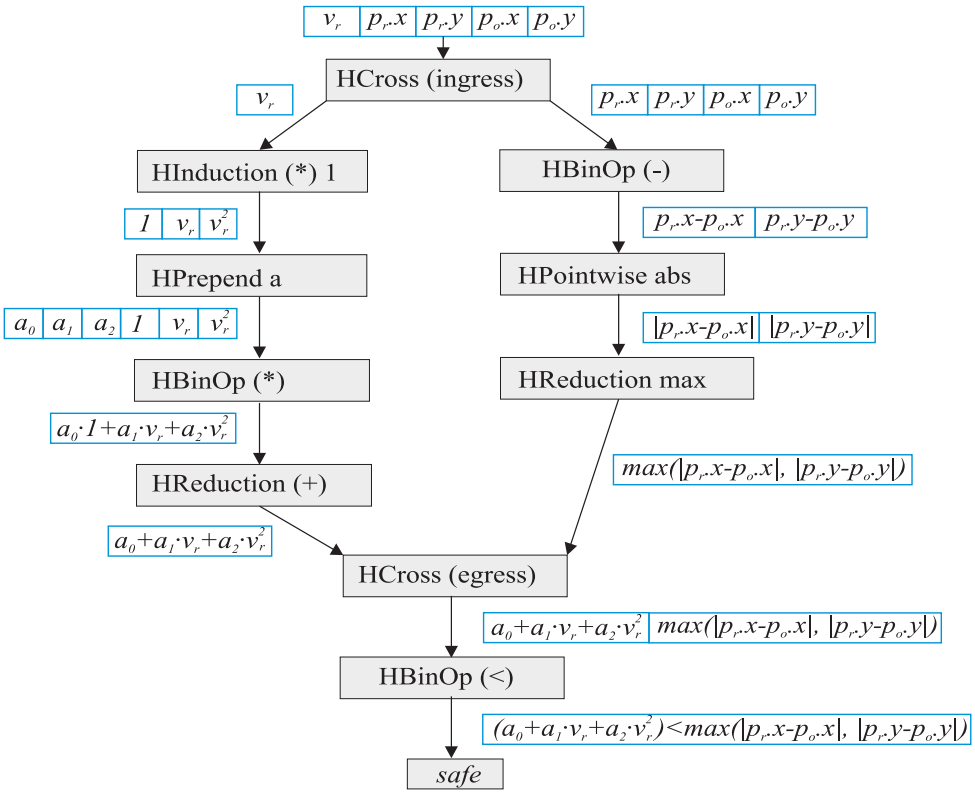

Fig. 5. Dynamic Window Monitor dataflow graph in *HCOL*

2.2.1 *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.

2.2.2 `HScalarProd` *breakdown rule*

Here is an example of a SPIRAL breakdown rule and the corresponding lemma in HELIX for the `HScalarProd` operator. See Appendix A.2 for the description of all *HCOL* operators.

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 $\vec{x} = [x_0, x_1, \ldots, x_{n+n-1}]$, it computes $[x_0 \cdot x_n + x_1 \cdot x_{n+1} + \cdots + x_{n-1} \cdot x_{n+n-1}]$.

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 $[x_0 \cdot x_n, x_1 \cdot x_{n+1}, \ldots, x_{n-1} \cdot x_{n+n-1}]$. The operator takes as parameter f a function with type $\mathbb{I}_n \to \mathcal{R} \to \mathcal{R} \to \mathcal{R}$ 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 $[0 + x_0 \cdot x_n + x_1 \cdot x_{n+1} + \cdots + x_{n-1} \cdot x_{n+n-1}]$ as a right fold. The corresponding lemma and the definition of the helper function are shown in Listing 4.

```
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. `HScalarProd` breakdown rule

2.2.3 HCOL *semantics preservation verification framework*

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 𝓡 i → vector 𝓡 o) :=
    op_proper :> Proper ((=) ⟹ (=)) op.
```

Listing 5. `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 [76]. 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.

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

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 $h = h'$.

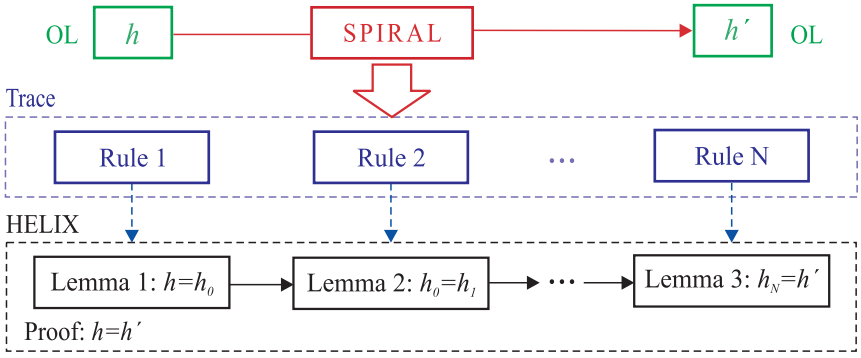

Fig. 6. HELIX translation validation of SPIRAL *HCOL* rewriting

This approach, shown in Figure 6, is an extension of the *translation validation* technique. 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.

3 Σ-*HCOL*

3.1 Σ-HCOL *language*

Most vector and matrix operations are naturally 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 7. 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.

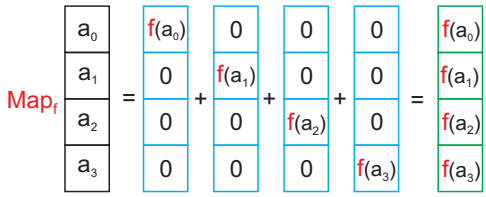

Fig. 7. Dense vector as sum of four sparse vectors

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 $f^2 : \mathbb{R}^2 \to \mathbb{R}^2$ which is implemented in C as **void** f2($\mathcal{R}$ src[2], $\mathcal{R}$ dst[2]), the sum would look like Figure 8.

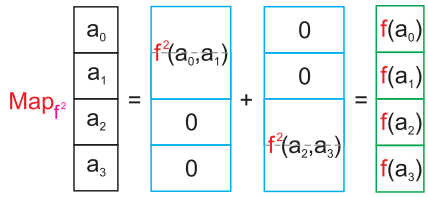

Fig. 8. Dense vector as sum of two sparse vectors

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 7, 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.

3.1.1 *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 in each pair 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, a well-formed Σ-*HCOL* expression triggers no collisions (see Section 3.1.5 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 the *Writer Monad* to track structural properties of carrier type $\mathcal{R}$, as described below.

```
Record Rflags : Type := mkRflags {is_struct: B; is_collision: 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* [57] library), we also need to define a *Monoid* record:

```
Definition mzero := mkRthetaFlags ⊤ ⊥.
Definition mappend (a b: Rflags) : Rflags :=
    mkRthetaFlags
      (is_struct a && is_struct b)
      (is_collision a || is_collision b ||
                    (negb (is_struct a || is_struct b))).
Definition Monoid_Rflags : Monoid Rflags := Build_Monoid mappend mzero.
```

Listing 6. Exclusive union monoid

The initial flags' value called *mempty* has the structural flag *true* and the collision flag *false*. The *mappend* operation shown in Listing 6 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 can be shown that the following Monoid laws are satisfied:

$$\forall a, \quad a \oplus 0 = 0 \oplus a = a \tag{2}$$

$$\forall a, \forall b, \forall c, \quad (a \oplus b) \oplus c = a \oplus (b \oplus c) \tag{3}$$

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 (2) and (3).

```
Instance MonoidLaws_Rflags: MonoidLaws Monoid_Rflags.
```

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 Rflags.
Definition Monad_Rflags := writer fm.
Definition Rfm := Monad_Rflags R.
```

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

```
Variable fm:Monoid Rflags.
Definition mkStruct (v:R) : Rfm fm := ret v.
Definition mkValue (v:R) : Rfm fm := tell (mkRflags ⊥ ⊥) ;; 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 6. We will call this type $\mathcal{R}_\theta$. A commonly used constant `0` 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 Rθ := Rfm Monoid_Rflags.
Definition 0 : Rθ := 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 $\mathcal{R}_\theta$. 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: R → R → R) (a b: Rθ),
  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: R → R → R) (a b: Rθ),
  evalWriter (liftM2 op a b) = op (evalWriter a) (evalWriter b).
```

The $\mathcal{R}_\theta$ is not the only parametrisation of $\mathcal{R}_{\text{fm}}$ 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 6 except with the *mappend'* function which tracks the preexisting collisions without generating new ones:

```
Definition mappend' (a b: Rflags) : Rflags :=
    mkRthetaFlags
      (is_struct a && is_struct b)
      (is_collision a || is_collision b).
Definition Monoid_Rflags' : Monoid Rflags :=
    Build_Monoid mappend' mzero.
```

Listing 7. "Safe" union monoid

This monoid instance allows us to define a "safe" variant of the $\mathcal{R}_\theta$ type as $\mathcal{R}_{\theta'} \triangleq \mathcal{R}_{\text{fm}}$ 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 $\mathcal{R}_\theta$ and $\mathcal{R}_{\theta'}$ is defined, which preserves both flags and values (see `SafeCast` and `UnSafeCast` operators in Appendix B.2).

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

Frequently, we need to combine vectors element-wise using the provided binary scalar function dot : $\mathcal{R} \to \mathcal{R} \to \mathcal{R}$. 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 {n:ℕ } (dot: ℛ → ℛ → ℛ)
: svector fm n → svector fm n → svector fm n.
```

Listing 8. `Vec2Union` vector combining operation

When `fm = Monoid_`$\mathcal{R}_{\text{flags}}$, this operation represents the combination of two partial computations. On the other hand, when `fm = 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 dot = $plus$, it is just vector addition.

3.1.2 *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).

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

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

3.1.3 *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:

$$\forall j < k, \quad {f_j}^{m \to n} : \mathbb{I}_m \to \mathbb{I}_n \tag{4}$$

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:

$$\forall a, \forall b, \forall i, \forall j, \quad f_a(i) = f_b(j) \implies (i = j) \wedge (a = b) \tag{5}$$

The family is called *surjective* if it satisfies:

$$\forall j, \exists a, \exists i, f_a(i) = j \tag{6}$$

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

The subscript indices in the mathematical notation used in formulas 4, 5, 6 are just additional arguments, and the actual type of the function is:

$$f : \mathbb{I}_k \to \mathbb{I}_m \to \mathbb{I}_n \tag{7}$$

Or in *uncurried* form:

$$f^{m \to n} : (\mathbb{I}_k \times \mathbb{I}_m) \to \mathbb{I}_n \tag{8}$$

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

3.1.4 Operator type

Σ-*HCOL* operators are defined using mixed embedding. By that, we mean that an operator's implementation is a 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 9.

```
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 9. `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.

3.1.5 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 `SHOperator_Facts` 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 `SHOperator_Facts`, it can be shown that all Σ-*HCOL* higher-order operators are also instances of `SHOperator_Facts`. That gives us a structural correctness proof "by construction" for any Σ-*HCOL* expression.

3.1.6 *Operator families*

Similar to families of index mapping functions, we can have families of operators. We define a *family* F of k operators as:

$$\forall \text{fm}, \forall i, \forall o, \forall \text{svalue}, \quad F : \mathbb{I}_k \to \text{SHOperator fm } i\ o\ \text{svalue} \tag{9}$$

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

3.1.7 *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 2.1.2. Here again, we use `Equiv` typeclass to define our equality relation for various types as described below.

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

```
Instance ℛfm_equiv: Equiv (ℛfm fm)) :=
  λ am bm ⇒ (evalWriter am) = (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 SHOperator_equiv {i o: ℕ} {svalue: ℛ}:
    Equiv (@SHOperator i o svalue) := λ a b ⇒ op a = 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*.

3.1.8 *Sparse embedding*

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

$$\texttt{SparseEmbedding}_{f,g,K} \triangleq (\lambda i.\, \texttt{Scat}_{f_i} \circ K_i \circ \texttt{Gath}_{g_i}) \tag{10}$$

The parameters are:

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

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 *gather* picks the input vector's elements, which are then processed by K, and the results are 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.

3.1.9 *Map-Reduce*

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

$$\mathbb{MR}_{k,f,z}\colon\ (\mathbb{N} \to (\mathcal{R}_{\_}^{n} \to \mathcal{R}_{\_}^{m})) \to \mathcal{R}_{\_}^{n} \to \mathcal{R}_{\_}^{m} \tag{11}$$

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 ($f : \mathcal{R} \to \mathcal{R} \to \mathcal{R}$). 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:

$$\mathbb{MR}_{2,+,0}(\lambda i.(\texttt{Scat}_{\lambda x.i} \circ [\![f]\!] \circ \texttt{Gath}_{\lambda x.i})) \tag{12}$$

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 (12) is shown in Figure 9.

3.1.10 *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*

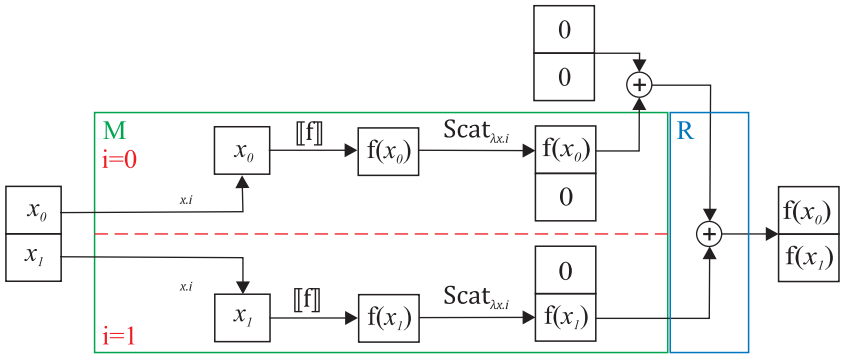

Fig. 9. Map-Reduce of a sparse embedding

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.

3.1.11 Running example

A result of the initial translation of our running example of an *HCOL* formulation of the dynamic window monitor from Listing 2 to Σ-*HCOL* is shown in Listing 10. It has been abridged to hide some non-essential parameters.

The full list of Σ-*HCOL* operators can be found in Appendix B.2.

```
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 10. Dynamic Window Monitor in Σ-*HCOL*

3.2 HCOL *to* Σ-HCOL *translation*

As mentioned in Section 3.1, *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 compostion 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 3.1.7).

Finally, we need to make sure that the resulting Σ-*HCOL* expression is *structurally correct*, as discussed in Section 3.1.5. 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 8.3.

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.

3.3 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 2.2.3.

3.3.1 *Running example*

The result of the Σ-*HCOL* rewriting steps of the expression from Listing 10 is shown in Listing 11

```
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 11. Dynamic Window Monitor in Σ-*HCOL* after rewriting

4 *MSHCOL*

4.1 MSHCOL *language family*

MSHCOL is an intermediate step in the HELIX transformation chain between purely functional Σ-*HCOL* and imperative *DHCOL* languages. Imperative language semantics, as is shown in Appendix D.3, 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, *MSHCOL* (M stands for *memory*). While *MSHCOL* 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 structural values. Besides physical data representation, the main change from Σ-*HCOL* is that we no longer 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 the introduction of implicit error handling in *MSHCOL*.

An example of both representations is shown in Figure 10. 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.

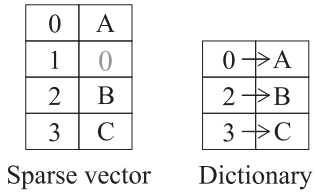

Fig. 10. Sparse vectors as dictionaries

Generally speaking, there is a 1-to-1 correspondence between Σ-*HCOL* and *MSHCOL* operators with the main difference in the input and output container data types. *MSHCOL* uses memory blocks, where Σ-*HCOL* uses sparse vectors. An example application of the `MApply2Union` operator in *MSHCOL* is shown in Figure 11. It is similar to the `Apply2Union` Σ-*HCOL* operator in Figure 32 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.

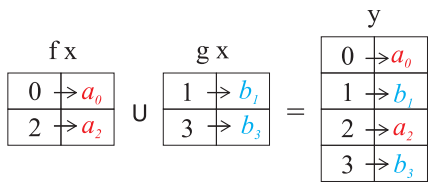

Fig. 11. `MApply2Union` in *MSHCOL*

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 *MSHCOL* operators could return errors and thus have the type: mem_block → option mem_block. However, we will prove later that our translation of a structurally correct Σ-*HCOL* program produces an *MSHCOL* program that does not err when applied to an input memory block that matches the expected input sparsity patterns.

In the title of this section, we have referred to *MSHCOL* as a "language family". The reason for this is it is implemented as a module, parametrized by the type of the data values. It is similar to $\mathcal{R}$ abstraction we used in *HCOL* and Σ-*HCOL*, but instead of typeclasses we are using here Coq's modules mechanism. We define `CType` module type which contains the following:

- Carrier Type: `CType.t`
- Constants: `Zero`, `One`.
- Equality: `Equiv`
- Operations: `Plus`, `Neg`, `Mult`, `Abs`, `Zless`, `Min`, `Max`, `Sub`.
- Inequality of constants: `Zero` $\neq$ `One`.
- Operations respect equality: $\forall$`op`, `Proper` ((=) $\Longrightarrow$ (=)) `op`.

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*. The MHCOL family consists of only one language, with `CType` instantiated for the set of real numbers $\mathbb{R}$. As we will see later in Section 5.6, `CType` can be also instantiated for IEEE floating point numbers.

Like in Σ-*HCOL*, we use *mixed embedding* [22] (a combination of *shallow* and *deep embedding*) to represent *MSHCOL* operators. We use `MSHOperator` record, which is indexed by dimensions $i, o \in \mathbb{N}$ of input and output memory blocks and contains the following:

- `mem_op`: `mem_block` $\rightarrow$ `option mem_block`
- `mem_op_proper`: `Proper` ((`equiv`) $\Longrightarrow$ (`equiv`)) `mem_op`
- `m_in_index_set`: $\mathbb{I}_i \rightarrow \mathbb{B}$
- `m_out_index_set`: $\mathbb{I}_o \rightarrow \mathbb{B}$

The fields include: a function `mem_op` implementing the operation on memory blocks which can fail (returning `None`); a witness for the function being a *proper morphism* [18] with respect to the setoid equality `equiv` (required because the carrier type is still abstract); and the two sets (represented as characteristic functions) which define input and output memory access patterns.

Additionally, all *MSHCOL* 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.

4.1.1 Memory model

MSHCOL definitions rely on *memory block* abstraction. In *MSHCOL*, 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 [47, 49] compiler 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. 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 12.

```
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 12. 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 13.

```
Definition mem_empty : mem_block.
Definition mem_lookup : ℕ→ mem_block → option CType.t.
Definition mem_add : ℕ→ CType.t → mem_block → mem_block.
Definition mem_delete : ℕ→ mem_block → mem_block.

Definition mem_in : ℕ→ mem_block → Prop.
Parameter decidable_mem_in : ∀ (k : ℕ) (m : mem_block), decidable (mem_in k m).
```

```
Definition mem_keys_lst : mem_block → list ℕ.
Definition mem_value_lst : mem_block → list CType.t.
```

Listing 13. 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.

Currently, we define only one language from *MSHCOL* language family: *MSHCOL*. This language uses type $\mathbb{R}$ to represent numerical values. It is implemented by using a `CType` instance for $\mathbb{R}$ to specialize *MSHCOL*. Up to this point, we kept the data type abstract enough to allow instantiating it with types like $\mathbb{R}$ or even $\mathbb{C}$. At this stage, we are intentionally narrowing it down to $\mathbb{R}$ in line with our intention to generate code working with floating-point machine numbers.

4.1.2 Running example

A result of a translation of the Σ-*HCOL* formulation of a dynamic window monitor from Listing 11 to *MSHCOL* is shown in Listing 14. 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)))
          (MSHIReduction 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))
          (MSHIReduction 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 14. Dynamic Window Monitor in *MSHCOL*

4.2 Σ-HCOL *to* MSHCOL *translation*

4.2.1 *Implementation*

Translation from Σ-*HCOL* to *MSHCOL* is implemented using the Coq meta-programming plugin, Template-Coq [77]. 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 *MSHCOL* 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 10 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 *MSHCOL* and mapped to corresponding binders in the resulting expression. Thus, the result of the *MSHCOL* translation of the dynamic window monitor Σ-*HCOL* definition from Listing 10 will also be parametrized by a and have the actual type:

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

4.2.2 *Proof of semantics preservation*

The semantic equivalence between an Σ-*HCOL* and an *MSHCOL* operator is defined as the `SH_MSH_Operator_compat` typeclass. It ensures that they have the same 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 15. Σ-*HCOL* and *MSHCOL* 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 *MSHCOL* 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. Assuming one-to-one operator correspondence, the syntax-driven proof automation applies manually proven per-operator `SH_MSH_Operator_compat` instances. This should succeed for all *MSHCOL* generated from the corresponding Σ-*HCOL* ones by our translation Template-Coq metaprogram.

5 *DHCOL* language family

DHCOL is a family of imperative languages, deep-embedded in Coq proof assistant. Languages in the family share their memory model with those of *MSHCOL*. In addition to dynamic memory, *DHCOL* has lexically scoped variables, which can hold `CType.t` values, pointers to memory blocks, and integers (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*.

It should be noted that *DHCOL* does not define general-purpose programming languages. It serves as an intermediate representation language for the class of problems that HELIX is designed for. As such, it primarily focuses on operations on $\mathcal{R}$-valued vectors. Only the features vital to represent corresponding HELIX abstractions are incorporated into the language. Given these design objectives, it may seem fairly esoteric compared to general-purpose programming languages.

The *DHCOL* memory model, shared with *MSHCOL*, is described in Section 4.1.1. 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 on integer values and `CType.t` (discussed in the next section) values, constant memory blocks, and pointers to memory blocks. `CType.t` and integer value expressions allow constants and provide a set of arithmetic operations such as addition, subtraction, and division. Additionally, expressions could reference variables from the environment, which are also typed. Evaluation of an expression has no side effects but could fail.

The sections 5.1 to 5.3 provide more technical details on the formalisation of *DHCOL* in Coq. The language syntax, list of operators and formal semantics are provided in Appendix D.

5.1 Type parametrization

Like the previous language family in the HELIX transformation chain, *DHCOL* also uses `CType` for data stored in memory addresses. Unlike *MSHCOL* however, *DHCOL* also defines a new type for integer values, `NType`. The primary reason for this is that with the languages of the *DHCOL* family we also transition to lower-level datatypes: from $\mathbb{R}$ to binary floats, and from $\mathbb{N}$ to machine integers. In fact, the difference in these types is the only thing separating languages of the *DHCOL* family from each other.

Our `NType` module type definition is similar to `CType` with a few additions:

- Constant: `zero`.
- Operations: `plus`, `minus`, `mult`, `div`, `mod`, `min`, `max`.
- Conversion to strings: `to_string`.
- Conversion to natural numbers: `to_`$\mathbb{N}$.
- Conversion from natural numbers: `from_`$\mathbb{N}$.

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.

A couple of properties must be proven for instances of this module type:

- $\forall x\ x_i\ z, \texttt{from\_}\mathbb{N}\ x = \texttt{inr}\ x_i \rightarrow y < x \rightarrow \exists y_i, \texttt{from\_}\mathbb{N}\ y = \texttt{inr}\ y_i$
- $\exists z, \texttt{from\_}\mathbb{N}\ 0 = \texttt{inr}\ z$

The first property states that if conversion from $\mathbb{N}$ succeeds for a given natural number, it must also succeed for all natural numbers below that one. This property could be considered a form of monotonicity. The second property simply states that the natural number 0 could always be successfully converted to `NTtype.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 implementation decision is meant to simplify proofs by induction.

5.2 *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 $\mathbb{N} \times$ 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 mem_block $\times$ NT.t.

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

Syntax of `AExpr` expressions is shown in Figure 12. The evaluation semantics for expressions will be discussed later in Section 5.3.

⟨AExpr⟩ ::= `AVar` ⟨Nat⟩ | `AConst` ⟨CTypeConst⟩ | `ANth` ⟨MExpr⟩ ⟨NExpr⟩ |
`AAbs` ⟨AExpr⟩ | `APlus` ⟨AExpr⟩ ⟨AExpr⟩ | `AMinus` ⟨AExpr⟩ ⟨AExpr⟩ |
`AMult` ⟨AExpr⟩ ⟨AExpr⟩ | `AZless` ⟨AExpr⟩ ⟨AExpr⟩ |
`AMin` ⟨AExpr⟩ ⟨AExpr⟩ | `AMax` ⟨AExpr⟩ ⟨AExpr⟩

Fig. 12. Syntax of `AExpr` expressions

Syntax for *DHCOL* operators is shown in Figure 13. In addition to the expressions shown in Figure 12, some *DHCOL* operators take `MemRef` values as arguments. `MemRef` value is a memory pointer combined with an offset as a natural number: `(PExpr, NExpr)`.

```
⟨DSHOperator⟩ ::= DSHNop |
                  DSHAssign ⟨MemRef⟩ ⟨MemRef⟩ |
                  DSHIMap ⟨Nat⟩ ⟨PExpr⟩ ⟨PExpr⟩ ⟨AExpr⟩ |
                  DSHBinOp ⟨Nat⟩ ⟨PExpr⟩ ⟨PExpr⟩ ⟨AExpr⟩ |
                  DSHMemMap2 ⟨Nat⟩ ⟨PExpr⟩ ⟨PExpr⟩ ⟨PExpr⟩ ⟨AExpr⟩ |
                  DSHPower ⟨NExpr⟩ ⟨MemRef⟩ ⟨MemRef⟩
                    ⟨AExpr⟩ ⟨CTypeConst⟩ |
                  DSHLoop ⟨Nat⟩ ⟨DSHOperator⟩ |
                  DSHAlloc ⟨NTypeConst⟩ ⟨DSHOperator⟩ |
                  DSHMemInit ⟨PExpr⟩ ⟨CTypeConst⟩ |
                  DSHSeq ⟨DSHOperator⟩ ⟨DSHOperator⟩
```

Fig. 13. Syntax of *DHCOL* operators

Example of a *DHCOL* operator, DSHIMap. Detailed description of all *DHCOL* operators can be found in Appendix D.2.

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]`.

5.3 Evaluation

DHCOL evaluation is performed within an *evaluation context*:

```
Inductive DSHVal :=
  | DSHnatVal (n:NT.t): DSHVal
  | DSHCTypeVal (a:CT.t): DSHVal
  | DSHPtrVal (a:ℕ) (size:NT.t): DSHVal.

Definition evalContext := list (DSHVal*𝔹).
```

It is a list where each position corresponds to the *de Bruijn* index of a variable and holds its value and *protection flag*. The function of the *protection flag* is to ensure that some variables, such as loop indices, are not accessible within loop bodies where they are not supposed to be exposed.

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 D.3.

5.4 RHCOL *language*

RHCOL is the first imperative language used in HELIX's compilation chain. It is the instantiation of the *DHCOL* family with real numbers $\mathbb{R}$ as the carrier type `CType.t` and natural numbers $\mathbb{N}$ as the integer type `NType.t`. The arithmetic operators on *CType* and *NType* are instantiated following standard mathematical definitions for $\mathbb{R}$ and $\mathbb{N}$ types. This means that no special handling of numerical computation is required at this step: iterators do not overflow, memory and pointer sizes are unlimited, and operations and comparisons are exact, among other things.

Running example

The result of a translation of the *MSHCOL* formulation of a dynamic window monitor from Listing 14 to *RHCOL* is shown in Listing 16. For simplicity, we use standard mathematical notation for arithmetic operations over `CType.t` and `NType.t`, array subscription notation for access to memory blocks and semicolon notation for `DSHSeq` operator within Listing 16.

```
Definition dynwin_FHCOL : DSHOperator :=
  DSHAlloc 2
    (DSHAlloc 1
       (DSHMemInit (PVar 0) CarrierAz;
        DSHAlloc 1
          (DSHLoop 3
             (DSHAlloc 1
                (PVar 7 [ NConst 0 ] = PVar 0 [ NConst 0 ];
                 DSHAlloc 1
                   (DSHPower
                      (NVar 2) (PVar 1, NConst 0) (PVar 0, NConst 0)
                      (AVar 1 * AVar 0) CarrierA1;
                    DSHIMap 1 (PVar 0) (PVar 3)
                              (AVar 0 * PVar 8 [ NVar 4 ])));
           DSHMemMap2 1 (PVar 2) (PVar 1)
                        (PVar 2) (AVar 1 + AVar 0))));
     PVar 0 [ NConst 0 ] = PVar 1 [ NConst 0 ];
     DSHAlloc 1
       (DSHMemInit (PVar 0) CarrierAz;
        DSHAlloc 1
          (DSHLoop 2
             (DSHAlloc 2
                (DSHLoop 2
                   (DSHAlloc 1
                      (PVar 9
                         [ NConst 1 + NVar 3 * NConst 1 +
                           NVar 1 * NConst 2 * NConst 1 ] =
                         PVar 0 [ NConst 0 ];
```

```
                    PVar 0 [ NConst 0 ] = PVar 2 [ NVar 1 ]));
                DSHBinOp 1 (PVar 0) (PVar 2)
                           (AAbs (AVar 1 - AVar 0)));
            DSHMemMap2 1 (PVar 2) (PVar 1) (PVar 2)
                         (AMax (AVar 1) (AVar 0))));
    PVar 0 [ NConst 0 ] = PVar 1 [ NConst 1 ]);
DSHBinOp 1 (PVar 0) (PVar 2) (AVar 1 < AVar 0)).
```

Listing 16. Dynamic Window Monitor in *DHCOL*

5.5 MSHCOL *to* RHCOL *translation*

5.5.1 *Implementation*

Translation from *MSHCOL* to *RHCOL* is also implemented in Template-Coq as a recursive descent on *MSHCOL*'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 *MSHCOL* operator which is a function with type `(mem_block → option mem_block)`, the compiler is parameterized by two memory pointer expressions specifying where an imperative *RHCOL* program corresponding to this function should read input from (`x_p`) and write output to (`y_p`).

The translation, if it succeeds, returns a *RHCOL* program and the list of "global" variables it depends on. Recall that *MSHCOL* operators are `MSHOperator` records, which may depend on additional parameters. All such additional parameters will be detected during translation to *RHCOL* and treated as "global" variables which must be present in the evaluation context prior to evaluating the *RHCOL* 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 *MSHCOL* operator is compiled to a *RHCOL* fragment, and the translation is compositional. For example, both `MSHEmbed` and `MSHPick` from *MSHCOL* translate to `DSHAssign` in *RHCOL*. Some *MSHCOL* operators translate not to a single *RHCOL* operator but rather to a *RHCOL* program fragment. Let us look more closely at an example of how `MSHCompose` is translated to *RHCOL*. The relevant part of the `match` statement on the *MSHCOL* operator's name and arguments from AST is shown in Listing 17:

```
    | Some n_SHCompose, [i1 ; o2 ; o3 ; op1 ; op2] ⇒
      ni1 ← tmUnquoteTyped ℕi1 ;;
      no2 ← tmUnquoteTyped ℕo2 ;;
      no3 ← tmUnquoteTyped ℕo3 ;;
      (* 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 17. `MSHCompose` operator translation to *RHCOL*

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

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

The variables in both *MSHCOL* AST and *RHCOL* 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 18.

```
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 18. Variable resolvers

The *var_resolver* is a map from *MSHCOL* to *RHCOL* 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 *RHCOL* introduces one or more new variables, which have no counterparts in *MSHCOL*. 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 17. 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 17, we "unquote"[5] 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 *RHCOL* variable indices,

[5]Template-Coq term which means decoding from AST representation.

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 *RHCOL* 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 $0 \ldots n-1$ representing the parameters of the function. While compiling a function, `Lambda_var_resolver` will map these indices as unchanged. However, when *MSHCOL* 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 *MSHCOL* 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 *MSHCOL* is 1. When compiling our lambda function, we use `(Lambda_var_resolver parent_resolver 1)`. When resolving i using this resolver, 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 14

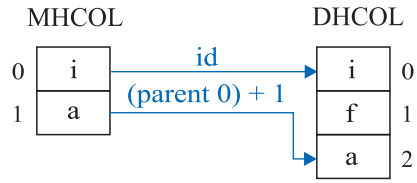

Fig. 14. Variable resolver example

5.5.2 *Proof of semantics preservation*

While the regular *MSHCOL* operators translate to a *RHCOL* program fragment, the higher-order operators translate into a sequence of instructions, with placeholders filled with *RHCOL* translations of their respective parameters. For example, *MSHCOL*'s `(MSHIReduction i o n z f op_family)` operator is compiled to the following *RHCOL* 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` operator 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 *RHCOL* correspond to `f` and `op_family` in *MSHCOL*, 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 *MSHCOL* to *RHCOL* 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 *MSHCOL* operator and its *RHCOL* translation. Then, these lemmas can be applied recursively, descending the structure of the reified *MSHCOL* expression hierarchically.

The first step in the process is to formalize the notion of semantic equivalence between a purely functional language with denotational semantics (*MSHCOL*) and an imperative language with operational semantics (*RHCOL*). Each *MSHCOL* operator is a function $x \mapsto y$ where `x` and `y` are memory blocks[6]. These functions are *pure functions* without side effects, whose output `y` depends on `x` and other variables in scope. On the other hand, a *RHCOL* translation of this *MSHCOL* 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 *RHCOL* programs representing pure functions is expressed as a `DSH_pure` typeclass:

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

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

Listing 19. `DSH_pure` typeclass

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 *MSHCOL* operator and a "pure" *RHCOL* program. Since the *MSHCOL* part of this relation is a function, we need to universally quantify on all possible inputs. Since *RHCOL* operators read and modify memory, the input and output of this function must correspond to some existing memory blocks. In *RHCOL* 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

[6] We omit error handling for now.

class parameterized by the respective *MSHCOL* and *RHCOL* operators, the evaluation context, and by the name of the input and output pointer variables in this context. Additionally, the purity of the *RHCOL* 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)));
      }.
```

Listing 20. `MSH_DSH_compat` typeclass

In the Listing 20, `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 $\mathtt{m}'$. Finally, the equality is reduced to the predicate `SHCOL_DSHCOL_mem_block_equiv` relating memory blocks `mb`, `ma`, and `md`.

Figure 15 shows the origin of these values in a case where no errors occur. Legend: σ is an evaluation context, and `m` and $\mathtt{m}'$ are memory states before and after execution of the `evalDSHOperator`. The `ma` corresponds to a memory block in $\mathtt{m}'$ referenced by `y_p`. The `md` is the result of applying the *MSHCOL* operator to `mx`.

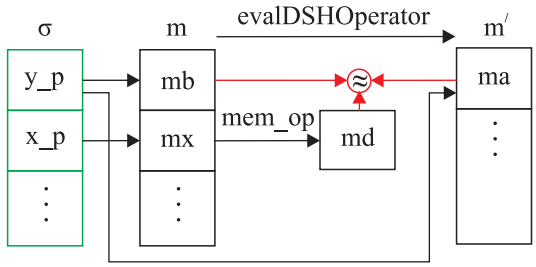

Fig. 15. *RHCOL* and *MSHCOL* equality relation

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 *MSHCOL* 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 *RHCOL*, 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 *RHCOL* execution
- `ma` - memory state of the output block after *RHCOL* execution
- `md` - values of changed output block elements after *MSHCOL* 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
:= ∀ 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 → b = a → MemOpDelta b a d
| MemExpected: is_Some d → a = d → 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 *MSHCOL* operators and their corresponding *RHCOL* equivalents, we can automatically generate proof for the result of any *MSHCOL* to *RHCOL* translation as an instance of this class for top-level *MSHCOL* and *RHCOL* expressions. During this proof automation, we need to recursively descend on an *MSHCOL* expression. The reason for this is that mapping between the two is not injective, and compiling two different *MSHCOL* operators could result in similar *RHCOL* constructs not easily distinguishable by simple matching on the structure. Whereas *MSHCOL* operators can be uniquely matched.

5.6 FHCOL *language*

The *FHCOL* language is an instantiation of *DHCOL* with machine numeric types. This brings us one abstraction step closer to machine code from *DHCOL*.

We instantiate `NType` module type with `MInt64asNT` module to represent unsigned 64-bit machine integers. The underlying machine integer (type `int`) type definition from the Vellvm library, which can be traced back to a similar CompCert definition, is defined as an arbitrary-precision integer (type `Z`) plus the proof that it is in the range from 0 (inclusive) to `modulus` (exclusive):

```
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 = 2^{64}. 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 the `CType` module type with the `MFloat64asCT` module, representing fixed-length 64-bit IEEE floating-point numbers. To work with IEEE floating-point numbers, we use the Flocq [13] library, which provides a comprehensive formalisation of IEEE floating-point arithmetic in Coq. Flocq supports a range of binary float representations. We are using Flocq's `binary64` type which corresponds to the `double` type in the C language. We instantiate the `CType` module type for the `binary64` type, which represents 64-bit IEEE floating-point numbers. As with integers, the definitions of all the required module fields are straightforward. Whenever the rounding mode needs to be specified, we always use *round to the nearest, ties to even*.

Running example

A result of the translation of a *RHCOL* formulation of the dynamic window monitor from Listing 16 to *FHCOL* is shown in Listing 21. For simplicity, we use standard mathematical notation for arithmetic operations over `CType.t` and `NType.t`, array subscription notation for access to memory blocks and semicolon notation for `DSHSeq` operator within Listing 21.

```
Definition dynwin_FHCOL : DSHOperator :=
  DSHAlloc 2%int64
    (DSHAlloc 1%int64
       (DSHMemInit (PVar 0) Float64asCT.Float64Zero;
        DSHAlloc 1%int64
          (DSHLoop 3
             (DSHAlloc 1%int64
                (PVar 7 [ NConst 0 ] = PVar 0 [ NConst 0 ];
                 DSHAlloc 1%int64
                   (DSHPower
                      (NVar 2) (PVar 1, NConst 0) (PVar 0, NConst 0)
                      (AVar 1 * AVar 0) Float64asCT.Float64One;
                    DSHIMap 1 (PVar 0) (PVar 3)
                            (AVar 0 * PVar 8 [ NVar 4 ])));
              DSHMemMap2 1 (PVar 2) (PVar 1)
                         (PVar 2) (AVar 1 + AVar 0))));
     PVar 0 [ NConst 0 ] = PVar 1 [ NConst 0 ];
     DSHAlloc 1%int64
       (DSHMemInit (PVar 0) Float64asCT.Float64Zero;
        DSHAlloc 1%int64
          (DSHLoop 2
             (DSHAlloc 2%int64
                (DSHLoop 2
                   (DSHAlloc 1%int64
                      (PVar 9
                         [ NConst 1 + NVar 3 * NConst 1 +
                           NVar 1 * NConst 2 * NConst 1 ] =
                         PVar 0 [ NConst 0 ];
                       PVar 0 [ NConst 0 ] = PVar 2 [ NVar 1 ]));
                 DSHBinOp 1 (PVar 0) (PVar 2)
                          (AAbs (AVar 1 - AVar 0)));
              DSHMemMap2 1 (PVar 2) (PVar 1) (PVar 2)
                         (AMax (AVar 1) (AVar 0))));
        PVar 0 [ NConst 0 ] = PVar 1 [ NConst 1 ]);
```

```
    DSHBinOp 1 (PVar 0) (PVar 2) (AVar 1 < AVar 0)).
```

Listing 21. Dynamic Window Monitor in *FHCOL*

5.7 RHCOL *to* FHCOL *translation*

5.7.1 *Implementation*

Translation between any two languages of the *DHCOL* family is straightforward, since they are implemented by different parametrizations of the same module and thus have the same syntax and semantics, up only to the primitive values and operations.

Translation is defined as a module parametrized by two *DHCOL* languages. We implement the translation function directly in Gallina. It works by recursively traversing the structure of `DSHOperator`, utilizing one-to-one correspondence between *DHCOL* and *FHCOL* language constructs.

```
Module MDHCOLTypeTranslator
       (Import CT: CType)
       (Import CT': CType)
       (Import NT: NType)
       (Import NT': NType)
       (Import L: MDSigmaHCOL(CT)(NT))
       (Import L': MDSigmaHCOL(CT')(NT'))
       (Import LE: MDSigmaHCOLEval(CT)(NT)(L))
       (Import LE': MDSigmaHCOLEval(CT')(NT')(L')).

[...]

Fixpoint translate (d: L.DSHOperator): err L'.DSHOperator := [...]
```

Listing 22. *DHCOL* family translator

The only reason a translation may fail is if a constant of the source language is not representable as a constant of the target language:

(1) Certain `CType.t` values may not be directly translatable. For example, during the *RHCOL* to *FHCOL* compilation pass, we might need to represent $\pi : \mathbb{R}$ exactly as a `binary64` number, which is not possible. 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 faces a similar issue. In the *RHCOL* to *FHCOL* translation case, values of type $\mathbb{N}$ need to be translated to `Int64`. The mapping is not total, as some natural numbers can not fit 64-bit 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.

5.7.2 *Proof of semantics preservation*

Semantic preservation is proven generically for a translation between any two *DHCOL* family languages. It is parametrized by *heterogeneous equivalence relations* on `CType.t`

and `NType.t` of the respective languages (Listing 23). The particular instantiation of these relations will influence resultant correctness guarantees: the more restricted the relation, the more restricted the guarantees.

This parametrization enables us to prove different statements for different translation passes. Most notably, an exact equivalence cannot be established for the $\mathbb{R}$ to `binary64` translation of the *RHCOL* to *FHCOL* step. Exactly equivalent real and floating-point numbers may not preserve this quality over equivalent operations due to rounding, preventing us from establishing such an invariant throughout the execution of respective programs.

For example, consider two real numbers, 0.1 and 0.2, and their sum 0.3, in real arithmetic. However, in IEEE 754 *binary64* (which corresponds to the `double` type in the C language) floating point arithmetic, $0.1 + 0.2 \neq 0.3$.

Thus for this step, instead of an exact relation, we use a trivial on which allows us to establish other necessary properties of the translation (such as the preservation of memory and stack operations), while offloading numerical analysis to be performed later. One example of such verified numerical analysis is shown in Section 7.

```
(* Heterogeneous equivalence on values before and after translation *)
Class NTranslation_heq :=
  {
    heq_NType : NT.t → NT'.t → Prop ;
    heq_NType_proper : Proper ((=) ⟹ (=) ⟹ iff) heq_NType;
    (* Value mapping should result in "equal" values *)
    translateNTypeConst_heq_NType :
      ∀ x x', translateNTypeConst x = inr x' → heq_NType x x';
  }.

Class CTranslation_heq :=
  {
    heq_CType' {env : Env} : CT.t → CT'.t → Prop ;
    heq_CType_proper : Proper ((≡) ⟹ (=) ⟹ (=) ⟹ iff) (@heq_CType');
    (* Value mapping should result in "equal" values *)
    translateCTypeConst_heq_CType :
      ∀ x x', translateCTypeConst x = inr x' →
            ∀ env, @heq_CType' env x x';
  }.
```

Listing 23. DHCOL primitive type equivalence classes

Proof of semantic preservation of the *RHCOL* to *FHCOL* translation also involves some numerical analysis, discussed below in sections 5.7.3 and 5.7.4.

5.7.3 *Floating-point numerical analysis*

The fact that the HELIX compilation chain targets efficient and portable (in LLVM IR) machine code, imposes a restriction on the set of data representations used by the final compiled program. In the compilation chain leading up to FHCOL, all representations of data and expressions have been either fully abstract ($\mathcal{R}$ in *HCOL* to *MSHCOL*) or infinitely precise ($\mathbb{R}$ in *RHCOL*).

Actual hardware usually compromises when representing real values and operations on them in favour of fixed representation size in memory and registers, and prioritises performance

over ultimate precision. As a consequence, fixed or floating-point types used by hardware cannot represent arbitrary real values exactly (in particular, irrational numbers, such as $\sqrt{2}$). Methods exist for achieving exact results using finite hardware, such as multiple-precision arithmetic, capable of perfectly capturing any computations on rational numbers; or symbolic computation which allows analysing abstract expressions over arbitrary variables. However, these are all much more resource-intensive than fixed precision arithmetic built into any processor; and so in HELIX we choose to use standard finite-precision IEEE-754 floating point arithmetic for operations in the compiled program.

This choice poses a problem for another stated goal of HELIX – safety. Almost any computation carried out in floating point arithmetic will necessarily accumulate an error that is only likely to grow, as more atomic operations are performed by the processor.

For most everyday applications, this imprecision is negligible. However, to provide safety guarantees on mission-critical systems HELIX is designed for, it must be quantified and analyzed. An unaccounted-for rounding error can lead to catastrophic results [9].

At this point, it is worth noting that the HELIX chain does not include any optimizations aimed at improving the numerical stability of the compiled algorithm.

In fact, HELIX also does not provide any general guarantees relating the numerical stability of the compiled program to that of the source expression (treated in a floating-point paradigm). *HCOL* breakdown (Section 2.2) and Σ-*HCOL* rewriting (Section 3.3) steps of the chain, as well as some of the compilation passes, while validated to be perfectly semantically preserving under exact arithmetic assumed by `CarrierProperties`, are not analyzed for rounding errors they might introduce at later steps.

Establishing a numeric boundary on the rounding error introduced by a particular program can not, by itself, be sufficient to prove that the program is safe for use in the real world. No constant error bound ϵ is universally safe for all applications; this part of the analysis necessarily requires understanding the physical meaning of the numbers.

For example, numerical analysis can establish that the rounding error in an algorithm does not exceed 0.001. But in order to establish that this number is safe, we need to first understand whether it measures millimeters or kilometers, as well as what tolerances are involved in our application – is it acceptable to miss by a meter?

For mission critical applications where the user may wish to verify the numerical properties of their algorithm, HELIX provides a foundation for statically analyzing error bounds on the outputs and proving custom safety properties of programs.

The mechanism for dealing with floating-point inaccuracy in HELIX is a symbolic execution engine. The output of any *FHCOL* program generated by HELIX can be provably reduced to a set of arithmetic expressions over that program's respective inputs.

Certain characteristics of HELIX-generated programs mean they lend themselves well to analysis via symbolic execution; some of the biggest challenges commonly associated with the approach can be overcome:

- *Path explosion.* While for general-purpose languages symbolic execution requires keeping track of many possible branches of execution, often resulting in exponential growth of the search space, HELIX-generated *FHCOL* programs are known to always take the same path over the execution tree. This means that a single pass is sufficient to explore the entire search space, or, in other words, that any *FHCOL* program corresponds to a single pre-defined set of arithmetic expressions.

- *Environment interactions.* Any HELIX-generated FHCOL program is pure, meaning no external calls need to be modeled.
- *Modeling memory, symbolic store updates.* The modular architecture of *DHCOL* enables symbolic execution of any program to be defined in terms of *DHCOL*'s regular semantics. In particular, it is sufficient to instantiate the Carrier Type (see Section 5.1) as the type of arithmetic S-expressions, to produce effectively a "symbolic-valued" language. This way, the language's memory itself becomes the equivalent of a symbolic store, and the regular execution procedure can be seen as a symbolic execution pass.

  Any *DHCOL*-family language can be converted into such a model using an instance of the standard translator (described in Section 5.7.1), and since all translations within *DHCOL* are proven to produce semantically equivalent programs, we can infer that the output of any *DHCOL* program is exactly the output of its symbolic counterpart, evaluated with the program's concrete input.

Combined with automated numerical analysis tools such as Gappa [25], this means that error bounds can be inferred automatically for all outputs of all *FHCOL* programs. See Section 7.3.1 for a case study on how it may be used in practice.

5.7.4 *Integer range analysis*

Several *DHCOL* operators, such as `DSHIMap`, `DHSBinOp` and `DSHLoop` are parameterised by a natural number, which controls the number of iterations during iterative computations and is usually related to the input vectors' dimensions. During their evaluation, the index's current value, upper-bounded by this parameter, will be assigned to an environment variable of type `NType.t`, which could be used in integer expressions.

While *RHCOL* maps `NType.t` to $\mathbb{N}$, in *FHCOL* it is mapped to unsigned fixed-width machine integers. This makes the transition from the unbounded $\mathbb{N}$ in *RHCOL* to bounded machine integers in *FHCOL* problematic as it is susceptible to an integer overflow. Additionally, evaluation of `NType.t` expressions in *RHCOL* and *FHCOL* might result with an integer underflow, which is handled differently within these languages. In *RHCOL* result of $a - b$ where $b > a$ is defined to be 0, while in *FHCOL* the result would be computed modulo 2^n.

Integer overflow and underflow can lead to incompatible behavior between an *RHCOL* and an *FHCOL* program, which might result in runtime errors such as out-of-bounds vector indices or wrong numbers of loop iterations, etc. The only way to guarantee the correct execution of a program is to prove that no such incompatible behavior is introduced during *RHCOL* to *FHCOL* translation. This property is not necessarily true of all HELIX-generated programs, but can still be automatically inferred for a given one, assuming it does hold.

A practical consideration worth mentioning here 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 and vector sizes represented as $\mathbb{N}$ in *RHCOL* are usually linked to data dimensions, that also makes them relatively small, and the overflow when translating into *FHCOL* is unlikely to occur.

The general approach for ensuring integer bounds guarantees for *RHCOL* to *FHCOL* translation is as follows: the main observation is that the structure of computation will be the same in both languages. At each step of the evaluation, there is a one-to-one relation

between their corresponding evaluation environments. For this analysis, we do not concern ourselves with `CType.T` variables and only consider variables holding `NType.t` values. We aim to prove that an integer overflow never occurs and that for all $\mathbb{N}$ variables in *RHCOL* are represented exactly as fixed-width integers in *FHCOL*.

However, ensuring the exact representation of integer variables is not enough. During the evaluation of iterative operators such as `DSHIMap`, the index will be used in integer expressions (`NExpr` type) which could exhibit an integer overflow during evaluation. For that reason, we need to ensure that the evaluation of such expressions, with all index values in the expected range, will not cause incompatible behavior due to an integer overflow or underflow.

In order to establish equivalence between *RHCOL* and *FHCOL* programs, we define *closure trace* semantics for *DHCOL* programs. Firstly, we consider an evaluation context that contains ranges of values for integer variables. Pointers and Carrier Type values in the context are ignored by replacing them with placeholders to preserve the de Bruijn indices. Listing 24 shows the type of values that are stored in the context. `DSHIndex n` denotes an integer variable in the range $[0, n]$, while `DSHOtherVar` is used as a placeholder for non-integer variables.

```
Inductive DSHIndexRange : Type :=
| DSHIndex : NT.t →DSHIndexRange
| DSHOtherVar: DSHIndexRange.
```

Listing 24. Reduced `DSHVal`

We use the term *range closure* to mean a pair consisting of an `NExpr` and an integer evaluation context with `DSHIndexRange` values for all `NType.t` variables it may reference.

While computing the closure trace semantics no numeric computations of any kind (not even integer operations) are performed. Instead, any time an integer expression would normally be evaluated, it is instead saved along with the ranges of its inputs as a ranged closure. Evaluating an operator then results in a list of all such closures, which we refer to as a *closure trace*. *DHCOL*'s lack of branching and statically-known loop bounds and vector sizes allow to compute the closure trace statically. The same trace will always be encountered for the same operator, independent of input. For example, during the evaluation of `DSHIMap` and `DSHBinOp` operators, we require the value of the variable used to access the index of the current iteration to be within the input vector size and during the evaluation of the `DSHLoop` operator we require the value of the variable that contains the loop index to be within the loop bounds.

Secondly, as shown in the Listing 25, we consider two range closures (before and after *RHCOL* to *FHCOL* translation) to be equivalent if their corresponding integer expressions n and n' are structurally equivalent (`heq_NExpr`) and evaluate (`evalNExpr`) to equivalent (`heq_NType`) results under all equivalent (`heq_evalContext`) contexts which are compatible with the range restrictions (`evalContext_in_range`).

Specifically, for *RHCOL* and *FHCOL* pairs, the `heq_NType` predicate relates natural numbers to fixed-width machine integers and defines that the second argument converted to $\mathbb{N}$ should match the first argument exactly. This ensures that no incompatible behavior occurs due to integer overflow or underflow during the evaluation of a `NExpr` expression.

```
Definition evalNExpr_closure_equiv
  '((σn, n) : LE.evalNatClosure)
  '((σn', n') : LE'.evalNatClosure)
```

```
  : Prop :=
  ∀ σ σ',
    LE.evalContext_in_range σ σn →
    LE'.evalContext_in_range σ' σn' →
    heq_evalContext σ σ' →
    heq_NExpr n n'
    →
    herr_c heq_NType (LE.evalNExpr σ n) (LE'.evalNExpr σ' n').
```

Listing 25. Equivalence of ranged closures

Finally, two operators (before and after translation) are equivalent if all range closures from their closure traces are equivalent. The full semantical preservation statement is proven by relying on this assumption. For a given operator, it becomes a proof obligation, discharged using simple proof automation (reducing the goal to a set of linear inequalities for all the ranges).

Listing 26 shows an example of a *DHCOL* program. In this example, DSHIMap operator is used inside DSHLoop operator. During the evaluation of both of these operators, an additional variable used to access the current iteration index is added to the context. This program is equivalent to the C program shown in Listing 27.

```
DSHLoop 3
  (DSHIMap 3
     (PVar 1) (PVar 2)
     (APlus (ANth (MPtrDeref (PVar 3)) (NVar 1))
            (ANth (MPtrDeref (PVar 5))
                  (NMult (NConst 3) (NVar 2))))).
```

Listing 26. Example of a DHCOL program

```
for (uint64_t i = 0; i < 3; i++)
    for (uint64_t j = 0; j < 3; j++)
        y[i] = x[i] + a[3 * j];
```

Listing 27. Equivalent C program

Listing 28 shows the computed closure trace for this program. In this example, the DSHIMap index variable (NVar 1, which corresponds to j in the C program) is bounded by the number of iterations that DSHIMap performs. For the DSHLoop operator, range closures are computed individually for each iteration: on every iteration, the loop index variable (NVar 2, which corresponds to i in the C program) is bounded by the index of the current iteration.

```
[(* loop iteration 3 *)
([DSHOtherVar; DSHIndex 3; DSHIndex 2; DSHOtherVar; DSHOtherVar; DSHOtherVar],
 NVar 1); (* i *)
([DSHOtherVar; DSHIndex 3; DSHIndex 2; DSHOtherVar; DSHOtherVar; DSHOtherVar],
 NMult (NConst 3) (NVar 2)); (* 3*j *)
 (* loop iteration 2 *)
([DSHOtherVar; DSHIndex 3; DSHIndex 1; DSHOtherVar; DSHOtherVar; DSHOtherVar],
 NVar 1); (* i *)
([DSHOtherVar; DSHIndex 3; DSHIndex 1; DSHOtherVar; DSHOtherVar; DSHOtherVar],
 NMult (NConst 3) (NVar 2)); (* 3*j *)
 (* loop iteration 1 *)
([DSHOtherVar; DSHIndex 3; DSHIndex 0; DSHOtherVar; DSHOtherVar; DSHOtherVar],
 NVar 1); (* i *)
```

```
([DSHOtherVar; DSHIndex 3; DSHIndex 0; DSHOtherVar; DSHOtherVar; DSHOtherVar],
NMult (NConst 3) (NVar 2)) (* 3*j *)]
```

Listing 28. Closure trace of a DHCOL program

The example program contains two integer expressions that correspond to i and (3 ∗ j), which are used to access values from the input vector x and the constant parameter a respectively. Thus, the resulting closure trace contains six range closures: range closures for the two integer expressions are computed for each of the three loop iterations.

6 LLVM IR

We are reaching the final step in our chain: *FHCOL* programs are further compiled to LLVM IR. 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 18.1.1 supports code generation for IA-32, x86-64, ARM, Qualcomm Hexagon, LoongArch, M68K, MIPS, NVIDIA Parallel Thread Execution, PowerPC, AMD TeraScale, SPARC, z/Architecture, and XCore. Using LLVM toolchain terminology, our *FHCOL* to IR compiler can be considered an LLVM *front end* for the *FHCOL* language.[7]

This compilation step is notable as the LLVM IR is a general-purpose language not specifically designed for this project. In particular, it is the first Turing-complete language in HELIX compilation chain. As such, the gap to bridge is arguably larger than in the previous step, especially in terms of semantic preservation.

At a high level, LLVM IR programs are sets of mutually recursive functions, where functions are control flow graphs with blocks of code that define local variables following the static single assignment (SSA) invariant. We refer the interested reader to the reference manual[8] for an extensive description of the language.

To build our verified compilation chain, we naturally need a formal version of LLVM IR: HELIX relies to this end on the Vellvm project [89, 98, 99]. Vellvm exposes a formal language, dubbed *VIR*, deeply embedded in Coq, that models a large fragment of LLVM IR. Its infrastructure offers (1) a parser for valid LLVM IR syntax into its internal representation of (VIR) programs, (2) a formal semantics equipped with a battery of lemmas to reason about refinement of programs, (3) a pretty printer into valid LLVM IR syntax for LLVM toolchain interoperability, and (4) an executable interpreter proven correct with respect to *VIR* semantics, for testing purposes.

Its semantics is relatively unconventional: programs are modelled as monadic computations. Intuitively and to a first approximation, one can think of it as writing a monadic interpreter in a language like Haskell. The major caveat, however, is that unlike Haskell, our host language, Gallina, does not allow for divergence. This issue, which may seem as a showstopper, is resolved by representing programs as coinductive objects, elements to a generalization of Capretta's delay monad [17]: no divergence is built internally to Coq, and a lazy interpreter is recovered by extraction to OCaml. This semantics is built using a modern library providing a wealth of semantic tools for building and reasoning about such monadic interpreters, the *Interaction Trees* (ITrees) library [84, 88].

[7] Conversely, from the HELIX point of view, the compiler can be considered a *back end*.
[8] https://llvm.org/docs/LangRef.html

Verifying the translation of *FHCOL* to *VIR* constitutes one of the largest stress test to Vellvm's meta-theory, as well as more generally to the itree-based approach to verified compilation. Through this section, we aim to describe the technique, but also document the lessons for both of these projects we believe have emerged from the experience.

We first focus on the description of the compilation pass itself, before discussing the formal theorems we establish, and major elements of their proof.

6.1 *Compiling from* FHCOL *to* VIR

6.1.1 *Type mapping*

Both *FHCOL* and LLVM use IEEE-754 64-bit binary floats, and during compilation, *FHCOL*'s `CType.t` type is mapped to `TYPE_Double` in *VIR*'s AST. Both languages use Flocq's `binary64` type for floating-point values.

FHCOL's `NType.t` is mapped to `TYPE_I64` in *VIR*'s AST. Similarly, *FHCOL*'s `Int64.int` values are mapped to *VIR*'s own `int` type, which implements 64-bit unsigned integers. 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 operations used. For instance, the `udiv` instruction will perform unsigned integer division, while the `sdiv` instruction performs signed division. Since *FHCOL* integers are always non-negative, we will always emit unsigned instructions when generating IR code.

FHCOL Memory blocks are mapped to IR memory regions and accessed using pointers to `[n x double]` where n is the 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.

6.1.2 *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 29.

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

Listing 29. *FHCOL* Program definition

The record contains the name of the function to generate, a list of global variable names with their corresponding types, and the `DSHOperator`, which represents the program code.

To recall from 5.3, *FHCOL* evaluation context is a De-Bruijn-indexed list of typed variables. In a well-formed *FHCOL* program, the `op` assumes that the evaluation context contains global variables starting from index 0 and matching the order and types from `globals`, followed by two variables which are pointers to allocated memory blocks of sizes i and o, holding the input and output data vectors, respectively.

All global *FHCOL* variables are mapped to corresponding *VIR* global variables, which, depending on the compiler invocation flags, are declared with either the *external* or *internal* linkage type. Specifically, when compiler flags indicate code generation for testing, global

variables are initialized with random data, as discussed in Section 8.1. Local *FHCOL* variables are similarly mapped to *VIR* local variables.

6.1.3 Compiler organisation

The *FHCOL* to *VIR* compiler is written in Gallina. It translates *FHCOL* programs into a corresponding *VIR* AST. 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").

```
(* List of blocks with entry point *)
Definition segment:Type := block_id * list (block typ).

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 compiled program 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 30

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

Listing 30. Compiler state

The types of all *FHCOL* variables are recorded in Γ since *FHCOL* does not distinguish between global and local variables as they both live the in evaluation context. When compiling the top-level *FHCOL* operator, all pre-defined variables are assumed to be *global* in *VIR* while all new variables created when opening scopes (e.g. memory pointers in `DSHAlloc` or loop indices in `DSHLoop`) are considered *local* in *VIR*.

The counters in `IRState` are used to generate unique identifiers for blocks, local identifiers, and void identifiers.[9] 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.

In contrast, global identifiers already have pre-assigned names in `FSHCOLProgram.globals`, and these names are used in the generated *VIR* code. It is assumed that they do not use the same prefix as locally generated variables, and the compiler performs additional checks to ensure they are unique and do not collide with the names of built-in LLVM intrinsics.

Since *FHCOL* is lexically scoped, as the compiler proceeds with structural recursion over the *FHCOL* structure, Γ at each *FHCOL* variable's de Bruijn index holds the LLVM global variable names or compiler-assigned register names and their *VIR* 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.

To give an example of internal compiler workings, we will now discuss how loops with integer indices are compiled. Such loops are used to compile *FHCOL* operators such as `DSHIMap`, `DSHBinOp`, `DSHMemMap2`, and others. The generated *VIR* code should follow 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
```

[9] *void* identifiers are used in Vellvm's SSA form to assign results of *VIR* operators, such as `store`, which do not return any values.

```
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 abstracted into 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.

6.2 Verification: background

6.2.1 A primer on Interaction Trees

Interaction Trees [82] (ITrees) have emerged in the Coq ecosystem as a rich toolbox for building compositional and modular monadic interpreters for first order languages. ITrees are first and foremost a data structure for representing computations interacting with an external environment through *visible events*. Concretely, ITrees are defined as:

```
CoInductive itree (E: Type → Type) (R: Type) : Type :=
  | Ret (r: R)          (* pure computation *)
  | Tau (t: itree E R)  (* "silent" tau transition *)
  | Vis {A: Type} (e : E A) (k : A → itree E R). (* event e yielding an answer in A *)
```

The datatype takes two parameters: a signature `E` that specifies the set of interactions the computation may have with the environment, and the type `R` of values that it may return. ITree computations can be thought of as trees built out of three constructors. Leaves, via the `Ret` constructor, model pure computations, carrying `R` values. `Vis` nodes model an effect `e` being performed, before yielding to the continuation `k` with the value resulting from `e`. The `Tau` constructor represents a non-observable internal step that occurs. Notably, ITrees are defined coinductively, allowing them to model diverging computations as non-well-founded trees.

ITrees form a convenient model of computation. They are a monad, supporting the usual `(ret : A → itree E A)` operation to embed pure computations into the monad, and the sequencing of computations via `(bind (c : itree E A)(k : A → itree E B): itree E B)`. But they furthermore allow for modeling loops and recursion. In particular, the `(iter (body : I → itree E (I + A))(i : i): itree E A)` combinator builds the operation computing first `(body i)`, and then pursuing based on the value returned: if it is a new accumulator (a value `(inl j)`), it reenters the body, if it is a final result (a value `(inr a)`), it terminates on this value.

These operations come with the expected laws: `bind` is associative, `ret` acts as a unit on both side for the former, and `iter` satisfies the slightly more involved but unsurprising laws capturing the behavior of a loop.[10] The precise meaning of these equations depends on the notion of equality — generally ITrees are considered equivalent up to weak bisimulation. Trees co-terminate, i.e. both diverge or both terminate on the same value, while additionally having the same set of traces of external events.

Weak bisimilarity captures an adequate notion of equality over computations. For many purposes, however, logical equality on return values is too restrictive. For instance, computations which build memory states may need to be compared with a more extensional equivalence relation. More generally when reasoning about optimization, arbitrary relations are quickly necessary in order to emulate something akin to Benton's relational program logic [6]. The interaction tree library provides such facilities, specifically, `eutt R t u` expresses that the computations `t` and `u` coterminate, with the same traces of external events, and the returned value in each leaf is related by `R`. Crucially, this extension makes it possible to compare heterogeneous computations, in particular, to relate distinct memory models: *FHCOL*'s and *VIR*'s.

ITrees are *free* in the sense that external events can be given a concrete model into any monad,[11] i.e., can be implemented after the fact. This property is embodied in the `interp` function: given any *handler* `(h: ∀ X, E X → M X)` implementing the events `E` in the monad `M`, `interp h : ∀ X, itree E X → M X` lifts this implementation to computations.

Based on this notion of interpretation, the ITree library is designed to structure the semantics of languages by first representing their syntax as trees whose events abstract all effects, and implement these effects into monads progressively. The approach has proven to scale to realistic language, as we will see for instance in the case of Vellvm in Section 6.2.2. We illustrate here the idea on a minimal example.

Example. As a mean to illustrate, let us imagine we model a simple imperative language, with commands including assignments, sequencing and loops:

$$c ::= skip \mid c; c \mid x := e \mid \texttt{while}\ b\ \texttt{do}\ c$$

A first perspective on modelling commands is to represent them as elements of `itree MemE unit` where `MemE` describes read and write operations. The representation is built *by recursion on the structure of commands*: the difficult case, the `while` loop, is resolved thanks to the `iter` combinator. However, at this stage, equality of computations, defined as bisimilarity of the trees representing the commands, is too tight: the memory model is left unspecified. Expected equations, such as two consecutive writes to the same variables being equivalent

[10] I.e., the associated Kleisli category is traced.

[11] More precisely, any iterative monad: the monad must be able to represent diverging computations.

to only performing the second one, are not satisfied. Indeed, the bisimilarity observes the trace of all reads and writes that occur during the computation—reading twice from the same variable is different from reading only once. We hence refine the model by providing an implementation of reads and writes in a simple state monad: by *interpreting* the previous representation, we obtain a model of commands in `StateT mem (itree void)unit`, that is stateful computations that may diverge silently, or eventually return a final memory. On this model, pointwise bisimilarity provides a suitable notion of equivalence of programs.

6.2.2 A primer on Vellvm

Interaction Trees have been put to work in a variety of contexts: the verification of a web server [45, 96], of transactional objects [50], as semantic foundations for contextual refinement [74], and even RoboChart [86]. They perhaps find their largest application to date as part of the Vellvm [89] project: a realistic sequential fragment of LLVM IR is given a denotational semantics based on Interaction Trees.

The version of *VIR* against which we interface in this work is essentially based on the one presented in [89], largely extended with new contributions to its meta-theory. It covers most features of the core sequential fragment of LLVM IR 11.0.0 as per its informal specification, including: the basic operations on 1-, 8-, 32-, and 64-bit integers, `Doubles`, `Floats`, structs, arrays, pointers, and casts; `undef` and `poison`; SSA-structured control-flow-graphs, global data, mutually-recursive functions, and support for intrinsics. The main features that are currently unsupported are: some block terminators (`switch`, `resume`, indirect branching, `invoke`), the `landing_pad` and `va_arg` instructions, architecture-specific floats and opaque types. The list of supported intrinsics is small, but user-extensible.

At a bird's eye view, the denotation process is exactly the same as in the case of the toy imperative language from the previous section: we represent, by structural recursion on the syntax, programs into trees filled with interactions, and then plug a monadic interpretation for these interactions into the model. At a more technical level, things get significantly more complex: we highlight two aspects of this complexity.

First, the representation of *VIR* programs must operate at two distinct levels: from the perspective of a single function, and from the perspective of a complete program. In order to represent a *VIR* function, that is a control flow graph, one needs simply to define the semantics of jumps: following a denotational approach, they are resolved using the `iter` combinator—intuitively, one can think of a jump at the end of a block as a tail recursive call. However, a complete LLVM IR program is a set of statically known mutually recursive functions. The function-level representation therefore exposes function calls as external events, and the top-level representation resolves the mutual recursion. The interplay between these two perspectives is source of a significant overhead in reasoning.

Second, the list of effects a *VIR* program can exhibit goes far beyond simple read and writes. Rather, one needs to consider read/writes to global variables, read/writes to registers, interactions with the memory (including allocation and casts between pointers and integers), manipulation of the call stack, calls to intrinsics, non-deterministic access to under-defined values, and undefined behaviors. While we refer the interested reader to [89] for details, the situation is summed up graphically on Figure 16. Rather than a simple monolithic implementation of the effects as in the toy example described above, a stack of interpreters is incrementally layered atop of the initial representation of programs, as described in more

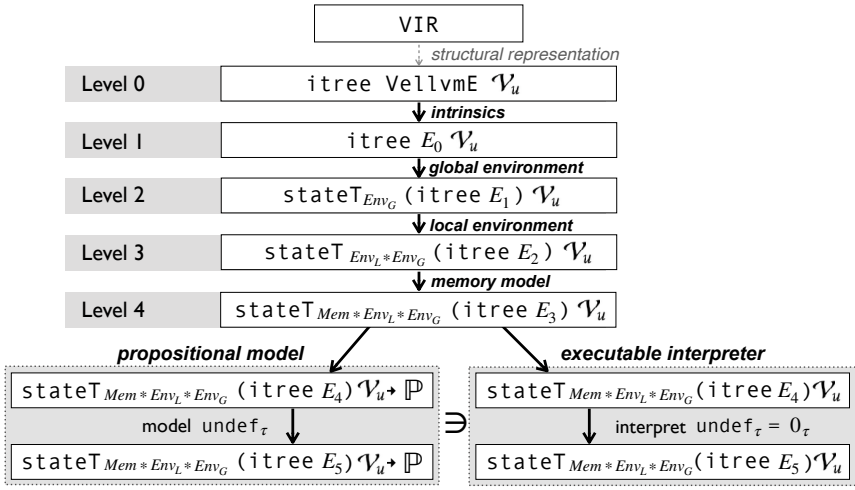

Fig. 16. Vellvm: Levels of interpretation

details by [88]. The fork in the stack happens where non-determinism comes into play: the semantics itself (the left path) captures sets of valid computations, while a deterministic, executable, alternative (the right path) is provided for testing purposes, and proved against the semantics to be a valid execution.

Crucially, each level of the layer induces its own notion of refinement of programs, and the refinements get increasingly coarser. We shall exploit this property in the present development: since the compiler from *FHCOL* to *VIR* does not introduce any non-deterministic effect, we are able to "pretend" that the semantics is deterministic by reasoning w.r.t. programs interpreted only until before the fork.

6.2.3 *Setting the scene: an itree-based semantics for* FHCOL

Interaction Trees offer a very flexible framework for establishing refinements of programs, including between heterogeneous computations operating over distinct memory models. However, they do assume both semantics of interest to be modeled using ITrees: before getting started, we hence need a bit of glue.

We provide an itree-based model for *FHCOL*. For simplicity, we stay as close as possible to the original fuel-based big step interpreter, but in a monadic style in the `itree Event` monad where `Event`. The interface `Event` is defined as the sum `MemEvent + StaticFailE + DynamicFailE`. We hence 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, such as accessing an undefined variable, which causes *DHCOL* compilation to fail. In contrast, *dynamic errors* refer to errors that cannot 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.
```

The `MemEvent` interface captures *FHCOL*-level memory event (read, writes, allocations and free):

```
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 31. DHCOL events

We define an event handler which maps memory events to the state monad on HELIX memory extended with dynamic and static failures:

```
Definition Mem_handler:
MemEvent ~> Monads.stateT memory (itree (StaticFailE +' DynamicFailE))
:= λ T e mem ⇒
match e with
| MemLU msg id ⇒
lift_Derr (Functor.fmap (λ  x ⇒ (mem,x)) (memory_lookup_err msg mem id))
| MemSet id blk ⇒ ret (memory_set mem id blk, tt)
| MemAlloc size ⇒ ret (mem, memory_next_key mem)
| MemFree id    ⇒ 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. The events describe the operations of allocation and de-allocation of memory blocks as well as reading and writing them.

To remain close to the interpreter, the evaluation context is passed around as an explicit argument to the representation function `denoteDSHOperator : evalContext → DSHOperator → itree Event unit`.

Naturally, to connect this new model to the upper layers of the compiler, we prove it sound against the big step semantics. I.e.:

```
Theorem Denote_Eval_Equiv:
  ∀ (s: evalContext) (op: DSHOperator) (mem: memory) (fuel: ℕ) (mem': memory),
  evalDSHOperator s op mem fuel = Some (inr mem') →
  eutt equiv (interp_Mem (denoteDSHOperator s op) mem) (Ret (mem', tt)).
```

Listing 32. The monadic model for *FHCOL* is sound against the interpreter

We consider any *FHCOL* operator `op` and any pair of evaluation context `s` and memory `mem` such that the big step semantics succeeds with final memory `mem'`. Then the monadic model of `op` initialized with `s` and `mem` is weakly bisimilar to a pure computation returning `mem'`. To be perfectly precise, we establish that the final memory is only extensionally equivalent to `mem'` (using the `equiv` argument given to `eutt`) so that we do not have to rely on functional

extensionality. This technicality can be soundly ignored. Finally, we write `interp_helix` for the composition of `interp_Mem` with an interpreter sending dynamic failure into the failure monad.

We can now abstract away from the big step semantics, and dedicate our attention to the problem of establishing the soundness of the compilation of *FHCOL* operators by relating their respective monadic models.

6.3 Verification: proof technique

The verification of a compiler for a general purpose language seeks to establish that the set of observable behaviors for the compiled program is included in the set of behaviors for the source program. In order to be termination-sensitive—to prove that diverging programs are compiled to diverging code—behaviors must include infinite traces, whether silent or reactive.

A standard proof method to establish such inclusion of behavior is the one popularized notably by CompCert. Semantics are assumed to be formalised as labelled transition systems, and elementary simulation diagrams are established for each program transformation considered. An isolated, generic, coinductive argument stitches these diagrams together to derive the inclusion of behavior.

This inclusion is established in a different manner for denotational semantics built upon ITrees, though the result is semantically equivalent. Users directly establish weak bisimilarities defined as a coinductive predicate—the `eutt` relation mentioned above. Of course this relation only relate uninterpreted computations, i.e. trees: one lifts this relation to the monadic domain at hand by, typically, quantifying over states.

Through this subsection, we highlight the important elements of proof methodology underlying the approach.

6.3.1 A relational program logic for termination and observation sensitive refinements of programs

Xia et al. introduced `eutt` as a way to define a notion of equivalence of computations modeled as ITrees. Here, we emphasize that, especially once the trees get interpreted into a state monad transformer, this relation benefits from being read as the fundamental building block for a relational program logic for termination-and-trace sensitive refinements.

Consider for instance the type of the denotation for open *VIR* control flow graphs, before introduction of the non-deterministic effects of the language: $[\![cfg]\!]_{\mathsf{cfg}} : \mathtt{M_V} \to \mathtt{L_V} \to \mathtt{G_V} \to \mathtt{itree}\ \mathtt{E_V}\ ((\mathtt{M_V} \times \mathtt{L_V} \times \mathtt{G_V}) \times ((bid \times bid) + \mathcal{V}_u)$. Similarly, consider the type of the denotation of *FHCOL* programs: $[\![\mathtt{op}]\!]^{\sigma}_{\mathsf{H}} : \mathtt{M_H} \to \mathtt{itree}\ \emptyset\ \mathtt{option}(\mathtt{M_H} \times \mathtt{unit})$, where $\emptyset$ denotes the empty signature. Building a refinement between *FHCOL* operators and *VIR* cfgs amounts to assuming that initial memories m_H and m_V are appropriately related by the precondition P (which also has access to *VIR* local and global state), proving the computations are bisimilar, and expressing as a postcondition Q the relation between the resulting states, as well as the dynamic value computed on the *VIR* side.

The relational program logic nature is further emphasized by the following notation:

$$\{P\}(op, cfg)\{Q\} \triangleq \forall\ m_H\ m_V\ l\ g,\ P\ (m_H, (m_V, (l, g))) \to [\![op]\!]_{\mathsf{H}} m_H \approx_Q [\![cfg]\!]_{\mathsf{cfg}}\ m_V\ l\ g.$$

$$\frac{R(r_1, r_2)}{\texttt{ret}\ r_1 \approx_R \texttt{ret}\ r_2}\ \textsc{ERet} \qquad \frac{t_1 \approx_U t_2 \qquad \forall\, u_1, u_2, U(u_1, u_2) \Rightarrow (k_1\ u_1) \approx_R (k_2\ u_2)}{(x \leftarrow t_1;; (k_1\ x)) \approx_R (x \leftarrow t_2;; (k_2\ x))}\ \textsc{EBind}$$

$$\frac{t_1 \approx_{R_1} t_2 \quad R_1 \subseteq R_2}{t_1 \approx_{R_2} t_2}\ \textsc{EMon} \qquad \frac{t_1 \approx_{R_1} t_2 \quad t_1 \approx_{R_2} t_2}{t_1 \approx_{R_1 \sqcap R_2} t_2}\ \textsc{EAnd} \qquad \frac{t_1 \approx t_1' \quad t_1' \approx_R t_2' \quad t_2' \approx t_2}{t_1 \approx_R t_2}\ \textsc{EuttCong}$$

Fig. 17. Relational reasoning principles

While this notation provides the right intuition, we keep things more shallow in our formal development as our language of assertions itself is shallow, and as we need to work at various levels of denotation whose types differ slightly.

Figure 17 describes the main relational rules we need at the level of interaction trees to conduct the proof. Pure computations are related if they return values related by the postcondition. The EBind rule acts as a cut, or a traditional sequence rule: one can introduce any intermediate relation acting as a postcondition for the first part of the computations, and precondition for the following part. Note that in our case of interest, we apply this rule after having provided our stateful computation with its initial state, the return type therefore encompasses both the pure values computed, as well as the intermediate states. Separately established postconditions can be combined via the usual intersection of relations in EAnd[12]. The logic supports the expected weakening rule EMon: it is crucial in order to allow compositional reasoning. Finally, the $\approx$ relation is a congruence on either side of a refinement proof: we exploit this fact to compute symbolically during the refinement proofs.

6.3.2 *A symbolic interpreter for* VIR

Zakowski et al. introduced with *VIR* [89] how the equational theory supported by the ITree combinators—particularly the iterator used to resolve jumps–could be used to derive rich reasoning principles about open control flow graphs. We further commit to this approach by characterizing equationally the behavior of all syntactic constructs of the language and sealing off the denotation functions.

This proof theory of *VIR* can be organized into three main groups. For readability, we write these proof rules as inference rules: under the conditions above the line, the (possibly existentially quantified) equation below the line holds.

Structural rules, described in Figure 18, are mostly administrative, yet crucial: they spell out the compositionality of the semantics. The semantics of each syntactic construct can be unfolded in terms of the semantics of its sub-components, allowing for inductive hypotheses and refinements of smaller syntactic components to be invoked during the proof.

Figure 19 describes the rules governing the semantics of open control flow graphs in terms of its sub-graphs. Most of these rules are conditioned by static preconditions expressing notions of separation between the sets of input and output labels of the sub-graphs at play. The `independent` predicate asserts that all labels present in each graphs are disjoint, while `wf_cfg` ensures that all bound labels are unique. These lemmas are sufficiently expressive to reason about all cfgs built by the compiler, including the complex parametric looping construct described in Section 6 which plugs a body represented as an arbitrary open control flow graph inside of another one performing the iteration.

[12]It is worth noting that we lose this rule when we move into the non-deterministic monad.

$$\frac{}{[\![[]]\!]_c \approx \texttt{ret}\ tt}\ \textsc{CNil} \qquad \frac{}{[\![i :: c]\!]_c \approx [\![i]\!]_i\ ;;\ [\![c]\!]_c}\ \textsc{CCons} \qquad \frac{}{[\![c1 +\!\!+ c2]\!]_c \approx [\![c1]\!]_c\ ;;\ [\![c2]\!]_c}\ \textsc{CodeApp}$$

$$\frac{}{[\![\{b, \Phi_s, c, t\}]\!]_{bk}^{b_f} \approx [\![\Phi_s]\!]_{\Phi_s}^{b_f}\ ;;\ [\![c]\!]_c\ ;;\ [\![t]\!]_t}\ \textsc{BkUnfold} \qquad \frac{bks[b_s] = \texttt{None}}{[\![bks]\!]_{\texttt{bks}}(b_f, b_s) \approx \texttt{Ret inl}\ (b_f, b_s)}\ \textsc{Out}$$

$$\frac{bks[b_s] = \texttt{Some}\ bk}{[\![bks]\!]_{\texttt{bks}}(b_f, b_s) \approx fov \leftarrow [\![bk]\!]_{bk}^{b_f}\ ;;\ \texttt{tryret}\ fov\ \texttt{in}\ [\![bks]\!]_{\texttt{bks}}}\ \textsc{In}$$

$$\texttt{tryret}\ fov\ \texttt{in}\ f \triangleq \texttt{match}\ fov\ \texttt{with} \mid \texttt{inl}\ x \Rightarrow f(x) \mid \texttt{inr}\ v \Rightarrow \texttt{Ret}\ v$$

Fig. 18. Structural *VIR* equations (excerpt)

$$\frac{\texttt{outputs}(bks_2) \cap \texttt{inputs}(bks_1) = \emptyset \quad b_s \notin \texttt{inputs}(bks_1)}{[\![bks_1 +\!\!+ bks_2]\!]_{\texttt{bks}}\ (b_f, b_s) \approx [\![bks_2]\!]_{\texttt{bks}}\ (b_f, b_s)}\ \textsc{OcfgSeq1}$$

$$\frac{\texttt{outputs}(bks_2) \cap \texttt{inputs}(bks_1) = \emptyset}{[\![bks_1 +\!\!+ bks_2]\!]_{\texttt{bks}}\ (b_f, b_s) \approx fov \leftarrow [\![bks_1]\!]_{\texttt{bks}}\ (b_f, b_s)\ ;;\ \texttt{tryret}\ fov\ \texttt{in}\ [\![bks_2]\!]_{\texttt{bks}}}\ \textsc{OcfgSeq2}$$

$$\frac{\texttt{independent}\ bks_1\ bks_2 \quad b_s \in \texttt{inputs}(bks_1)}{[\![bks_1 +\!\!+ bks_2]\!]_{\texttt{bks}}(b_f, b_s) \approx [\![bks_1]\!]_{\texttt{bks}}(b_f, b_s)}\ \textsc{OcfgBrL}$$

$$\frac{\texttt{independent}\ bks_1\ bks_2 \quad b_s \in \texttt{inputs}(bks_2)}{[\![bks_1 +\!\!+ bks_2]\!]_{\texttt{bks}}(b_f, b_s) \approx [\![bks_2]\!]_{\texttt{bks}}(b_f, b_s)}\ \textsc{OcfgBrR}$$

$$\frac{bks = (prefix +\!\!+ bks' +\!\!+ postfix) \quad \texttt{wf\_cfg}(bks)}{[\![bks]\!]_{\texttt{bks}}\ (b_f, b_s) \approx fov \leftarrow [\![bks']\!]_{\texttt{bks}}\ (b_f, b_s)\ ;;\ \texttt{tryret}\ fov\ \texttt{in}\ [\![bks]\!]_{\texttt{bks}}}\ \textsc{OcfgSub}$$

Fig. 19. *VIR* proof rules for reasoning about open programs

Finally, we provide atomic semantic rules for *VIR* expressions, instructions, and terminators – a minimal excerpt is provided on Figure 20. These equations relate the denotation of atomic constructions to pure computations, i.e. of the shape `Ret` _. Because `Ret` is a left unit for `bind`, these rules symbolically reduce. For instance if i is an instruction, $[\![i]\!]_i\ g\ l\ m\ ;;\ k$, binds i's result in the continuation k. In contrast to the semantics itself, which is implemented as a computable total function in Gallina, these rules can be partial and rely on propositional preconditions.

We exploit this partiality in particular to provide a layer of abstraction when reasoning about *VIR*'s memory over a sane subset of the language. The memory model in all its generality is complex: it has to allow for accesses to arbitrary structured data, as well as pointer-to-int and int-to-pointer casts. These low-level memory features are implemented via a quasi-concrete memory model, strongly inspired by [43], but most of the complexities of this memory model (such as precisely how dynamic values are serialized into bytes) are irrelevant to the semantics of the reasonably well-behaved programs that *FHCOL* compiles down to. We're therefore able to provide higher level memory accesses (`read`, `write`, `allocated`,..) whose relative behavior is specified through lemmas. In particular when it comes to the kind of getelementptr instruction we generate, we provide the illusion of accessing array-like structures.

$$\frac{[\![e]\!]_e\ g\ l\ m \approx \texttt{Ret}\ (m, (l, (g, i)))\quad i \in \{0, 1\}}{[\![\texttt{br}\ b_0\ b_1]\!]_t\ g\ l\ m \approx \texttt{Ret}\ (m, (l, (g, \texttt{inl}\ b_i)))}\ \textsc{Br}$$

$$\frac{}{[\![\texttt{br1}\ b]\!]_t\ g\ l\ m \approx \texttt{Ret}\ (m, (l, (g, \texttt{inl}\ b)))}\ \textsc{Br1}$$

$$\frac{[\![op]\!]_e\ g\ l\ m \approx \texttt{Ret}\ (m, (l, (g, uv)))}{[\![id = op]\!]_i\ g\ l\ m \approx \texttt{Ret}\ (m, (l[id \leftarrow uv], (g, tt)))}\ \textsc{Op}$$

$$\frac{\texttt{size}(\tau) > 0}{\begin{array}{l}\exists m'\ a,\ \texttt{allocate}\ m\tau = \texttt{inr}\ (m', a) \wedge \\ \qquad [\![id = \texttt{alloca}\ \tau]\!]_i\ g\ l\ m \approx \texttt{Ret}\ (m', (l[id \leftarrow a], (g, tt)))\end{array}}\ \textsc{Al}$$

$$\frac{\begin{array}{l}[\![ptr]\!]_e\ g\ l\ m \approx \texttt{Ret}\ (m, (l, (g, a))) \\ [\![e_{ix}]\!]_e\ g\ l\ m \approx \texttt{Ret}\ (m, (l, (g, \texttt{nat2i64}(ix)))) \wedge \texttt{get\_cell}\ m\ a\ ix\ \tau = \texttt{inr}\ val\end{array}}{\begin{array}{ll}\exists ptr, & \texttt{read}\ m\ ptr\ \tau = \texttt{inr}\ val \wedge \\ & [\![id = gep\ [\tau]\ ptr\ [0_{i64}; e_{ix}]]\!]_i\ g\ l\ m \approx \texttt{Ret}\ (m, (l[id \leftarrow ptr], (g, tt)))\end{array}}\ \textsc{gepA}$$

Fig. 20. Specifications for *VIR* (excerpt)

All put together, this proof theory achieves several goals. First, it allows us to keep the representation functions and interpreters opaque, hiding their internal representation as itrees, recovering reasonably performant and convenient setoid-based rewrites when reasoning about *VIR*'s syntax. Second, it lifts the reasoning to the level of syntactic *VIR* program, i.e., yielding notably a more convenient way to provide abstract interfaces with the memory model. Finally, we define a symbolic interpreter for *VIR* that can be run *inside* of refinement proofs between *VIR* programs and arbitrary other computations.

All equations are indeed established with respect to $\approx$: they can be rewritten under $\approx_R$ for any arbitrary R. We hence provide a *reduction* tactic matching on a refinement goal to rewrite the adequate equations and clean up the goal: the denotation of a *VIR* syntactic component gets reduced into the `bind` of the denotation of an instruction, or a terminator, followed by its continuation. The atomic instruction at the head can then be symbolically *stepped* — we provide a partial tactic to perform this as well, but leave it in the hands of users. Naturally, this symbolic reduction can only be performed over concrete code, but it allows us to reduce code such as that generated by the compiler as far as possible: typical compiled components start with concrete code, followed by some parametric code resulting from the call to other compilation sub-routines.

6.4 Verification: compilation of operators

We now turn our focus to the `genIR` segment of the compilation pass. Recall from Section 6.1.3 that this function compiles an *FHCOL* operator to a *VIR open* control flow graph. More specifically, `genIR` outputs the *VIR* function implementing the compiled operator, save for a missing final return block to the main function that only gets inserted at the top-level, in the `LLVMGen` unit.

Monadic semantics built using the ITree framework can seamlessly express the semantics of open programs, such as the ones generated by `genIR`. We hence can focus on the correctness of the `genIR` compilation unit in isolation, without having to capture their syntactic context.

The statement we establish is, formally:

```
Theorem compile_FSHCOL_correct :
  ∀ (** Compiler bits *) (s1 s2: IRState)
    (** Helix bits   *) (op: DSHOperator) (σ : evalContext) (memH : memoryH)
    (** Vellvm bits  *) (nextblock bid_in bid_from : block_id) (bks : list block)
                        (g : global_env) (ρ : local_env) (memV : memoryV),
  (* (1) Evaluation succeeds *)
  no_failure (interp_helix (denoteDSHOperator σ op) memH) →
  (* (2) Compilation succeeds *)
  genIR op nextblock s1 = inr (s2,(bid_in,bks)) →
  (* (3) Preconditions on IRStates *)
  bid_bound s1 nextblock →
  Gamma_safe σ s1 s2 →
  (* (4) Memory invariant *)
  state_invariant σ s1 memH (memV, (ρ, g)) →
  (* (5) Equivalence of computations, and postconditions *)
  eutt (succ_cfg (genIR_post σ s1 s2 nextblock ρ g))
       (interp_helix (denoteDSHOperator σ op) memH)
       (interp_cfg  (denote_ocfg bks (bid_from,bid_in)) g ρ memV).
```

Listing 33. Correctness statement for `genIR`

This statement is a mouthful: we first unpack it informally, before delving formally into its components and proofs in the remainder of this subsection. It reads in English: if (1) for any *FHCOL* operator `op` and initial state from which evaluation succeeds, (2) assuming that the compilation of this operator succeeds, (3) if the `IRStates` satisfy the adequate preconditions, and (4) the initial memories are adequately related, then (5) the monadic denotations of the operator and compiled code are bisimilar, and produce adequately related results.

We now detail some technicalities hidden behind this theorem and its proof.

6.4.1 *On freshness of names, and well-formedness of graphs*

We focus first our attention on (3), the assumptions made on `IRStates`. Recall from Section 6.1.3 that an `IRState` s is the compiler state recording essentially two pieces of information: a collection of monotonic counters used to generate fresh names and an environment $s.\Gamma$ recording the type of the previously generated *VIR* identifiers. In the following, we pretend there is a single high watermark counter $s.n$ for simplicity.

From a programmatic side, this provides a compiler engineer with a convenient, simple abstraction: essentially a freshness monad with an additional log Γ to retrieve typing information. When it comes to reasoning about the compiler, invariants must be made explicit. Posing s_1 and s_2 as the respective `IRState` of the input and output of the compilation unit, these invariants can be abstracted as:

(1) the *VIR*-level typing information tracked in $s_1.\Gamma$ must soundly reflect the *FHCOL*-level evaluation context σ;
(2) variables recorded in $s_1.\Gamma$ should have been generated by counters below $s_1.n$;
(3) the `nextblock` passed to the compiler is generated by a counter below $s_1.n$;
(4) local variables introduced in the compilation of the operator must have been generated by counters comprised in the interval $[s_1.n, s_2.n]$—this fact, among others, is captured in the post-condition (5).

Lesson 1: Graph combinators for *VIR*

The effort of checking that the freshness monad and its use in writing the compiler are correct should be internalized inside a library. We conjecture that this could be achieved by a well-designed graph combinators library exposing a suitable DSL for writing frontends targeting *VIR*. The library should have the following characteristics:

- each combinator is characterized by a semantic equation;
- any graph built out of the surface DSL should be well-formed (in particular, in SSA form with no collusion of names) by construction;
- the set of combinators should be expressive enough to support writing compilers such as the one presented here.

We believe the design of this library to be a non-trivial but worthwhile endeavor.

This collection of facts ensures that our freshness monad actually yields fresh names, and that configurations are always well-formed. Finally, an additional memory invariant, elaborated upon in Section 6.4.4, relates the data stored in memory at the *FHCOL*-level to the data stored in registers and memory at the *VIR*-level.

A considerable amount of boilerplate and reasoning is necessary to maintain these invariants. Setting aside the memory invariant, the set of invariants necessary to formulate to capture the behavior of the freshness monad and the consequent well-formedness of the generated graphs is surprisingly intricate. Naturally, the effort to preserve them as part of the correctness proof for a compilation unit is on par. We had to develop over 600 lines of specification and 1,000 lines of proofs in isolation just to handle the simple arithmetic involved, ensuring that the compiler generates variables from counters within the range of its input/output freshness counters.

This experience leads us to draw Lesson 1: the need for a library of graph combinators.

6.4.2 On characterizing successful executions

Let us now consider a seemingly innocuous aspect of the theorem in Listing 33: Premise (1) states that the *FHCOL* operator considered should have a valid semantics. Hypotheses of this flavor are routine in the context of verified compilation, as notably embodied by the famous absence of undefined behavior assumption made by the CompCert compiler [48].

The adoption of an unconventional semantic model compels us to reevaluate commonplace approaches. When working with a small-step semantics, the assumption is typically expressed as a safety predicate: all reachable states, w.r.t. the transitive closure of our reduction, are safe, i.e., cannot trigger undefined behavior, or failure. In this setting, when proceeding by induction on the syntax of the source program, deriving this safety assumption for the intermediate states at play is straightforward: these states are reached by reduction, and hence are safe as well by definition.

We are here tasked with expressing the same semantic intuition, but in the context of a monadic computation. We have three fairly natural choices to draw from:

(1) explicit use the small step transition system denoting the monadic computation, and piggyback on the usual approach;
(2) leave dynamic failure in the computation as an external event, and craft the coinductive predicate stating the absence of these events from the tree;

Lesson 2: The image of a monadic computation is crucial to inverting equations between binds, and hence predicates in particular

While the `eutt` family of relation provides the adequate ingredient to derive a unary logic suitable to express invariants such as `no_failure`, great care must be taken to invert invariants over monadic sequences. The invariant can only be derived in the continuation at reachable points, as captures by the notion of image of a monadic computation in [88].

(3) interpret failure into the fail monad and express safety as a post-condition of the computation.

Making explicit the underlying transition system has been leveraged in similar context when reasoning on non-deterministic computations, notably by [19] on CTrees and [23] on DTrees. It is however a heavyweight device, especially for our sequential context. The two other options are in essence quite similar. Having experienced with both, we have chosen to favor (3) in order to reuse as much as possible of the existing infrastructure surrounding the `eutt` relation. Let us be more precise.

As observed in [88], one can derive a unary logic for ITree computations by simply considering the diagonal of the binary relation `eutt`. I.e., given a computation `c: itree E X` and a postcondition `Q : X → Prop`, `Q` is a postcondition for `c`, written `c [Q]`, if `eutt (λ x _ ⇒ Q x)c c`. With this definition, we can simply define the `no_failure t` predicate over computations in the failure monad transformer by ensuring in their postcondition they did not fail: `no_failure t := t [λ x ⇒ x <> None]`.

While the definition is trivial, its use has ramifications. Of course, the generic program logic defined over arbitrary ITrees is minimalist: one typically lifts it after stateful interpretation, and derive richer rules for the source language under consideration. However, it does inherit a generic cut rule from `eutt: c [S] → (∀ x, S x → k x [Q])→ c >>= k [Q]`.

This is however a *forward* proof rule. In order to manipulate `no_failure` during a proof, say to derive that the semantic of two subexpressions are safe given that the semantics of their addition is safe, one needs a backward rule, inverting a fact of the shape `c >>= k [Q]`. However, one can easily observe that the proof rule above is not complete. Consider `c = ret true` and `k = λ b ⇒ if b then ret 0 else ret 1`: the computation `c >>= k` satisfies the postcondition `λ x ⇒ x = 0`, but certainly `k` does not at all points.

This observation led [88] to introduce the notion of the image of a monadic computation in order to precisely capture the points in the continuation that are reachable, restricting in the previous example the safety in `k` to the `true` branch. We refer the interested reader to [88] for further details.

6.4.3 On the overall proof structure

The use of denotational techniques brings in two standard conveniences: (1) expressing the equivalence of open programs is straightforward and (2), we can reason by induction on the syntax despite the presence of general recursion.

These two principles guide the overall structure of the proof: for each translation unit of the compiler, we express its semantic correctness, and prove it by induction on the syntax of the component handled by this unit. These correctness lemmas culminate in `compile_FSHCOL_correct`, which is itself proven by induction on the operator `op`.

Lesson 3: Languaged-level reasoning principles are mandatory

Smaller developments based on ITrees have had the tendency to simply expose the model and reason over it directly. Once reaching the scale of proofs such as the one at hand here, this approach simply becomes intractable. Sealing off all denotation functions, as pointed out in Section 6.3.2, and building the combination of an equational theory and a relational program logic at the level of the source languages quickly becomes mandatory.
This experiment suggests the perspective of interfacing *VIR* with a modern logic such as Iris.

Each of these lemmas roughly follow the same pattern: in each case of the induction, terms on both side are executed symbolically—using the symbolic interpreter described in Section 6.3.2 on the *VIR* side, and the much simpler interpreter on the *FHCOL* side. When hitting a parameter, the cut rule combined with either an inductive proof, or a previously establish lemma for the translation unit, is used to make progress. Finally, the invariants are reestablished at the leaves.

6.4.4 *On the memory invariant*

Compositional reasoning techniques absolve us from maintaining complex invariants about the execution context. A well designed library of combinators, as suggested in Lesson 1 would go one step further in this direction. The heart of the job however remains: we need to reason about the invariants relating the states of the source and target languages. We give here an intuitive overview of the state invariant, referring to the formal development for its full complexity. We maintain the following facts: (1) the DHCOL and VIR memory state agree (2) the typing context is well-formed and the identifiers in the context are bound, (3) no pointers or identifiers are aliased, (4) all referenced identifiers are appropriately allocated, and (5) the static context is well-scoped.

The trickiest invariant is (1), the *memory invariant*, which describes how the dynamic values stored on the *FHCOL* side in a state σ are mapped into memory on the *VIR* side.

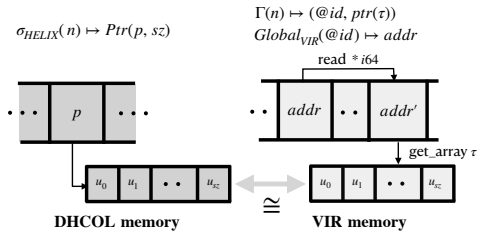

Fig. 21. Global pointer memory invariant

For each variable stored in σ, the invariant expresses how the static context Γ associates a *VIR* identifier storing in memory the same data—the details of this mapping depends on the type of data concerned. Figure 21 illustrates the situation for the case of a *FHCOL* variable containing a pointer to a vector stored in memory. The *VIR* identifier associated in Γ is global, storing (after a level of indirection) an address to memory such that reading at this address via

the *FHCOL*-specific getelementptr abstraction mentioned in Section 6.3.2 leads to decoding a list of dynamic values isomorphic to the source vector.

7 Dynamic Window Monitor

In Section 1.3, we have given a general overview of the *Dynamic Window Safety Monitor* approach and how its mathematical expression would be compiled using HELIX's verified chain. In the following section, we discuss in more detail the full extent of `DynWin`'s compilation chain and the end-to-end correctness guarantees that have been proven for it. It is intended as a mostly self-contained glossary of the techniques introduced earlier in the paper, all applied, in sequence, to a single example operator.

7.1 Overview

Let us begin by more closely examining the original arithmetic form of the `DynWin` expression and its inputs. Following [32], Equation (1) is first rewritten as the following system of equations, a form more suitable (for reasons discussed later in this section) for *HCOL* representation.

$$\text{safe}_{V,A,b,\epsilon} : \mathbb{R} \times \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{Z}_2; \tag{13}$$

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

$$d_\infty(\vec{x}, \vec{y}) \triangleq ||\vec{x} - \vec{y}||_\infty \tag{14}$$

$$p(x) \triangleq a_2 x^2 + a_1 x + a_0 \tag{15}$$

$$a_0 = \left(\frac{A}{b} + 1\right)\left(\frac{A}{2}\epsilon^2 + \epsilon V\right) \tag{16}$$

$$a_1 = \frac{V}{b} + \epsilon\left(\frac{A}{b} + 1\right) \tag{17}$$

$$a_2 = \frac{1}{2b} \tag{18}$$

$$\tag{19}$$

Table 2 lists all the input variables of the expression along with range constraints on them. The constraints have been selected to include most vehicle architectures on which the algorithm might be deployed (LandShark robot by Black-I Robotics and an American built car), while eliminating absurd situations such as faster-than-light travel. From an analytical point of view, this has two benefits: knowing that inputs are small allows us to automatically eliminate many concerns regarding floating-point overflow, while having tighter bounds on the numbers in turn greatly tightens the error bounds of output expressions at the numerical analysis step.

Informally, the inputs of the algorithm can be divided into two parts: the static (general problem-specific constants, the properties of the robot and its environment) and the dynamic (the information obtained by the robot during operation). We consider V, b, A and ϵ to be problem-specific constants, unchanging in any particular deployment of the algorithm. Notice also that in the breakdown (Equations (16) - (18)) the values of a_i only depend on these constants and can therefore be precomputed. The monitor function will take v_r, $p_r = (x_r, y_r)$,

[13] Same as 2011 BMW 520d.

Maximum obstacle velocity	$V \in [0, 20]\text{m/s}$	up to 72 km/h
Maximum robot acceleration	$A \in [0, 5]\text{m/s}^2$	0-60 mph in 8s[13]
Maximum robot breaking	$b \in [1, 6]\text{m/s}^2$	0.1-0.61g
Sampling period	$\epsilon \in [0.01, 0.1]\text{s}$	100-10 times per second
Robot longitudinal speed	$v_r \in [0, 20]\text{m/s}$	up to 72 km/h
Robot x coordinate	$x_r \in [-5000, 5000]\text{m}$	100 square km window
Robot y coordinate	$y_r \in [-5000, 5000]\text{m}$	100 square km window
Obstacle x coordinate	$x_o \in [-5000, 5000]\text{m}$	100 square km window
Obstacle y coordinate	$y_o \in [-5000, 5000]\text{m}$	100 square km window

Table 2. `DynWin` parameters and their ranges

and $p_o = (x_o, y_o)$ as inputs and return a Boolean value corresponding to the safety property (`True` standing for "safe to proceed", `False` - "unsafe, emergency stop required)".

7.2 *Unverified compilation with SPIRAL*

Running an *HCOL* expression from Listing 2 representing Equation (13) 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 the application of a sequence of semantically preserving transformation steps to the original expression to reshape it into its optimized form. The list of transformation steps is recorded in a "trace" file produced by SPIRAL. The produced C code will depend on parameters specific to the target platform. For the particular case where the target platform is set to generic x86 without SIMD instructions, the resulting C implementation of the dynamic monitor is shown in Listing 40. Its parameters, arrays `D` and `X` are expected to contain the vector a, and operating values of speed and position of the robot, respectively. Its return values are encoded as 1 for `True`, 0 for `False`.

For comparison, in Listing 41, we show the C code which is generated for an x86 platform with Intel Streaming SIMD Extensions (SSE) enabled. This version includes a runtime floating point error analysis using the *online uncertainty propagation* approach, as described in Section 5.7.3. In addition to 1 for `True` or 0 for `False`, this version could also return -1 to represent that the safety property could not be reliably computed for the given input values.

7.3 *Verified compilation with HELIX*

With HELIX, we begin by using the *HCOL* expression to encode the function from Equation (13), utilising shallow embedded syntax (`dynwin_orig` in Listing 2). The expression is parameterised by two vectors: one encodes the problem-specific constants a_i, and the other encodes the data received from the sensors. This expression is then processed through a series of verified transformation steps, as shown in Figure 2 in Section 1.2 and discussed below.

The first transformation applied to the original expression is HCOL *breakdown*. Based on the trace of optimizations obtained by running SPIRAL on a program equivalent to our *HCOL* input, the expression is transformed via a number of semantically preserving modifications. Currently, the sequence of steps is transferred from SPIRAL's trace manually, although

this process could easily be automated in the future. On the Coq side, each elementary transformation takes the form of a setoid rewrite, applied with a tactic, validated by HELIX.

This process results in an optimized *HCOL* representation of the program, `dynwin_HCOL`, displayed in Listing 3. At this stage, the soundness of the transformation means that the two Gallina functions have the same types and are extensionally equivalent. The only difference is that the new HCOL term is expected (by merit of SPIRAL's heuristics) to compile to more efficient machine code for the target architecture.

The next intermediate language of the chain of compilation is Σ-*HCOL*. Translating `dynwin_HCOL` to Σ-*HCOL* is a matter of "lifting" the *HCOL* expression, since any *HCOL* operator has a trivial corresponding Σ-*HCOL* version. This is done by wrapping the *HCOL* operator with `liftM_HOPerator`: the input, a sparse vector, is first "densified" - all structural flags are dropped and corresponding indices are filled with a default *svalue*, the *HCOL* operator is evaluated, and the result is "sparsified" by simply treating a dense vector as a sparse one with no structural elements.

However, such a translation is not useful on its own: the embedded *HCOL* expression would continue to operate on dense vectors, failing to make use of Σ-*HCOL*'s sparsity contracts and barring future optimizations making use of iterative computations. Once the operator is lifted (using `liftM_HOperator`), the lift function is further recursively distributed over the structure of `dynwin_HCOL` and individual constituent *HCOL* operations are replaced with their Σ-*HCOL* counterparts. Similar to the application of SPRIAL optimizations, this step is currently hardcoded, in the form of applying a series of rewriting rules to the operator. The result of this process, called `dynwin_SHCOL` is shown in Listing 10. Since lifting is proven to preserve outputs exactly, the resultant operator is known to be equivalent to its predecessor. Additionally, `dynwin_SHCOL` is proven to be structurally well-formed for parallelization by proving `SHOperator_Facts` class instance for it.

Once the initial Σ-*HCOL* operator is built, we can perform further optimizations inside it, this time making use of vector sparsity. Similarly to the HCOL *breakdown* step, we use SPIRAL's optimization trace as an oracle, only validating the results with HELIX. The final optimized Σ-*HCOL* expression, `dynwin_SHCOL1`, is shown in Listing 11.

The next step for `DynWin` is the *MSHCOL* language and the introduction of memory. The translation is fairly straightforward: using Template-Coq, the Σ-*HCOL* language operators are mapped (mostly) one-to-one into *MSHCOL* counterparts. The result (`dynwin_MHCOL`, shown in Listing 14) is structurally identical to the original Σ-*HCOL* operator, though, of course, with updated, memory-block-based semantics. Additionally, in this step, we specialize $\mathcal{R}$ as $\mathbb{R}$ by proving that $\mathbb{R}$ is indeed a legitimate specialization of $\mathcal{R}$ and poses all required properties.

With the introduction of memory blocks in *MSHCOL*, we shift toward an implicit model of environment and memory and an imperative paradigm in the languages of the *DHCOL* family. The chain goes through two of them: *RHCOL* and *FHCOL*.

First, with *RHCOL*, we transition to imperative code, while leaving the types unchanged. This is performed, as in the previous step, using Template-Coq. Unlike the Σ-*HCOL* to *MSHCOL* step, however, you may notice (see Listing 16) that the correspondence is far from one-to-one: a single *MSHCOL* function may compile into a construct of multiple nested *RHCOL* operators. In addition to returning the converted *RHCOL* operator, our MetaCoq compiler also generates a list of "global variables"; in the case of `DynWin`, this corresponds

to the vector a. The semantic preservation proof for this step is based on the assumption that precomputed values from a are already in memory before `DynWin` is executed.

The penultimate step, transition to *FHCOL*, is simple computationally: since both languages are instances of *DHCOL*, compilation between them is a trivial cast. The compiled *FHCOL* version of `DynWin` is syntactically exactly the same as the *RHCOL* one, the only difference is the types of constants.

Proving this transition correct however, poses a unique problem. As we have shown, in the compilation chain leading up to *FHCOL*, the outputs of all intermediate representations of the `DynWin` algorithm are exactly equivalent. At the *FHCOL* step, floating-point imprecision takes hold, and an exact computation is no longer achievable.

7.3.1 Numerical analysis

As discussed in Section 5.7.3, HELIX provides a mechanism for performing numerical analysis on *FHCOL* algorithms. Acceptable boundaries of precision vary by application, and thus the operator must make a choice on whether a particular compiled program is suitable for use, on a case-by-case basis.

In the case of `DynWin`, the output is always a single value, corresponding to one of two outcomes: "the robot is safe to proceed" or "the robot is not safe to proceed". We can use this, in conjunction with a degree of understanding of the underlying formula's physical meaning, to postulate a safety property which is sufficient for us to consider the *FHCOL* version of `DynWin` suitable for its purpose.

This property can be stated simply as follows: *if the infinitely precise version of DynWin deems the robot unsafe to proceed, the FHCOL version must also return "unsafe"*.

Let us elaborate. The mathematical model of robot safety, described in Equation 1, was proven correct in KeYmaera X [37]. This means that, interpreted with infinitely precise real-valued semantics, it can be treated as ground truth for evaluating the robot's situation in the physical world. We will call it the oracle. The algorithm we produce by compiling it with HELIX is different and could, in principle, produce an output that differes from the oracle for the same state of the robot. In table 3, we list all possible combinations of outputs from the oracle and the compiled algorithm at a given point in time.

No	Real world (oracle)	Computation (compiled algorithm)	Acceptable?
1	SAFE	SAFE	Yes: perfect sync
2	SAFE	UNSAFE	Yes: algorithm overly cautious
3	UNSAFE	SAFE	No: algorithm not cautious enough
4	UNSAFE	UNSAFE	Yes: perfect sync

Table 3. Possible situations arising at any point during the operation of the robot

Ideally, we would like only situations (1) and (4) to be possible - an algorithm in perfect sync with the oracle. As discussed however, this is not achievable in practice due to floating-point imprecision. We need to formulate a more permissive property that would still be sufficient to ensure the behavior of the robot guided by our program remains safe. One such property can be defined by including situation (2) from the Table 3 as acceptable: if the algorithm

classifies some safe situations as unsafe, the robot may sometimes stop unnecessarily[14], but so long as this is the only disagreement between the algorithm and the oracle (i.e. (3) does not occur), the robot will still keeps a safe course at all times.

In order to establish the discussed property, let us begin by calculating the error bounds on all floating-point computations performed by the `DynWin` *FHCOL* program.

The first step is to extract the arithmetic form of the computation from the full operator (see Listing 16). We can do so (as discussed in Section 5.7.3) by means of symbolic execution: translating the original operator to the "symbolic-valued" version of the language (named "*SFHCOL*"), executing it, and looking up the S-expression corresponding to the relevant part of the output in resultant memory. Listing 34 shows such a chain (the code is simplified, omitting some error handling for readability) and Listing 35 shows the computed S-expression. `(SVar i)` corresponds to a symbolic variable numbered i, while `dynwin_S_`σ and `dynwin_S_memory` designate an input environment filled with such arbitrarily (but uniquely) numbered variables.

```
Definition DynWin_S_out : option SFHCOL.memory :=
  (* translate into symbolic language *)
  DynWin_SFHCOL ← FHCOLtoSFHCOL.translate DynWin_FHCOL ;;
  (* evaluate with symbolic inputs *)
  s_m ← evalDSHOperator dynwin_S_σ DynWin_SFHCOL dynwin_S_memory ;;
  (* look up relevant block in memory *)
  s_ymb ← memory_lookup s_m dynwin_y_addr ;;
  (* look up relevant cell in block *)
  sexpr ← mem_lookup dynwin_y_offset s_ymb ;;
  ret sexpr.
```

Listing 34. DynWin symbolic execution

```
Some
  (SZLess
    (SPlus
      (SPlus (SPlus SConstZero (SMult SConstOne (SVar 0)))
             (SMult (SMult SConstOne (SVar 3)) (SVar 1)))
      (SMult (SMult (SMult SConstOne (SVar 3)) (SVar 3)) (SVar 2)))
    (SMax (SMax SConstZero (SAbs (SSub (SVar 4) (SVar 6))))
          (SAbs (SSub (SVar 5) (SVar 7)))))
```

Listing 35. DynWin symbolic execution result

Each of the variables in the input memory corresponds to some physical value (or an expression over such values) and has size constraints. In Listing 35, variables `(SVar 0)` through `(SVar 2)` correspond to elements of the vector a. For the scope of this problem, we assume values of a_i are precomputed using floating-point arithmetic. This is important, as the imprecision of this computation, while falling outside the focus area of our analysis (i.e. being performed outside the languages in the HELIX compilation chain), still impacts the precision of outputs obtained from compiled *FHCOL* programs.

To get around this issue, we require that a be computed using strictly IEEE-754 compliant arithmetic and exactly by the formulas 16, 17, 18 shown in Section 7.1. Relying on that, we

[14]How common such occurrence will be with our approach is discussed in more detail at the end of this section.

are able to infer the errors accumulated in a before it is passed to HELIX and use those in further analysis.

The first step of our analysis is to observe that the arithmetic form of `DynWin` obtained by symbolic execution is a single comparison: on the left-hand side is an expression corresponding to the polynomial (15), on the right-hand side - the Chebyshev distance (14) between the robot and the obstacle. With all inputs of these two expressions constrained, we can formulate a Gappa error analysis problem (see Listing 36 [15,16]).

```
@rnd64 = float<ieee_64, ne>;

poly = ((0.0 + (1.0*a0)) + ((1.0*v)*a1)) + (((1.0*v)*v)*a2);      # error-free lhs
poly64 rnd64 = ((0.0 + (1.0*a0)) + ((1.0*v)*a1)) + (((1.0*v)*v)*a2); # rounded lhs

cheb = x - y;                    # error-free rhs
cheb64 rnd64 = x - y;            # rounded rhs

{
  # input bounds
    a0 in [0, 12.15]
  ∧ a1 in [0.01, 20.6]
  ∧ a2 in [0.0833333, 0.5]
  ∧ v in [0, 20]
  ∧ x1 in [-5000, 5000]
  ∧ x2 in [-5000, 5000]
  →
    |poly64 - poly| in ?          # error in calculating lhs
  ∧ |cheb64 - cheb| in ?         # error in calculating rhs
}
```

Listing 36. DynWin polynomial error bounds

The resulting error bounds are 2×10^{-13} on the left-hand side and 9.1×10^{-13} on the right-hand side. We can interpret these as sizes of the intervals on the real line around infinitely precise results, in which floating-point results may fall.

Next, we notice that when trying to compare two numbers by comparing their floating-point representations, we can only get the wrong result if their respective error intervals intersect (Figure 22). We can make use of this: by restricting the comparison operator to only pairs of numbers separated by a sufficiently large interval (the "safety margin", denoted ϵ; see Figure 23), we can ensure that no two numbers are compared if their error intervals intersect. This behavior is equivalent to stating that any two numbers within ϵ of each other are "equal".

This approach - comparing floating-point numbers using some absolute or relative error margin, instead of relying on a direct comparison - is common practice in floating-point computations [24].

[15] Supplying Gappa with direct input is not necessary, all Gappa invocations can be performed from Ltac. The listing is provided purely for demonstration purposes.

[16] You may notice that Chebyshev distance is represented simply as subtraction in Gappa. This is because the other components of that computation, max and abs, are perfectly precise on floats and are not fully supported by Gappa.

In the case of absolute errors, a sufficient safety margin is the sum of error intervals of the two numbers being compared; for `DynWin` we have: $\epsilon = 2\times10^{-13}+9.1\times10^{-13} = 1.11\times10^{-12}$. The comparison operator is then implemented as shown in Listing 37.

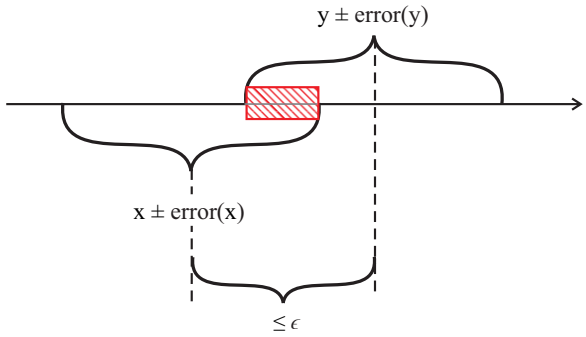

Fig. 22. Comparing two imprecise values can be unsafe if their error bounds intersect.

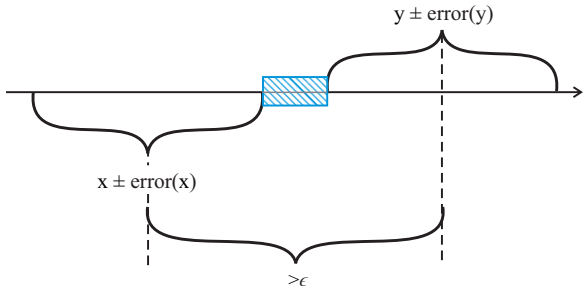

Fig. 23. It is safe to compare two imprecise values of sufficiently different magnitudes, as the error cannot have an influence on the comparison.

```
(* [a <? b = 1.0 ↔ b - a > epsilon] *)
Definition CTypeZLess (a b: binary64) : binary64 :=
  match Bcompare _ _ epsilon (CTypeSub b a) with
  | Some Datatypes.Lt ⇒ CTypeOne
  | _ ⇒ CTypeZero
  end.
```

Listing 37. Comparison operator with a safety margin

Expressed differently, such a comparison operator has an important property not possessed by the standard IEEE-754 comparison: if the two floats compare as `LT`, then any real-valued prototypes which they are approximations of must also compare as `LT`.

The introduction of a safety margin raises a question of false positives. After all, our definition theoretically allows excessively large choices of ϵ, which could, in practice, result in an unmoving, constantly trivially safe robot. We address this issue in Figure 24. Any invocation of the `DynWin` algorithm ultimately compares the Chebyshev distance between the robot and the obstacle to some number dependent on the robot's velocity $p(v)$; if the distance is smaller than $p(v)$, the robot is considered unsafe to proceed. On the robot's coordinate

plane this corresponds to an "unsafe area" in the shape of a square with side $2p(v)$ centered around the obstacle. Introducing an absolute safety margin as described, effectively means increasing $p(v)$ by ϵ. Thus, the unsafe area as computed by our floating-point algorithm, will be a square with side $2p(v) + 2\epsilon$ centered on the obstacle. In other words, the safety margin we introduced is a uniform border of thickness ϵ around the "actually unsafe" area. Considering the previously calculated value of $\epsilon = 1.11 \times 10^{-12}$, the safety margin we added is 0.00111 nm thick, which falls well within any practical limits for deployment of `DynWin`. Furthermore, it is worth nothing that even this margin could be tightened significantly, by basing the comparison operator on relative errors instead of absolute and by making more stringent requirements about the precision of precomputed values in the constant vector a.

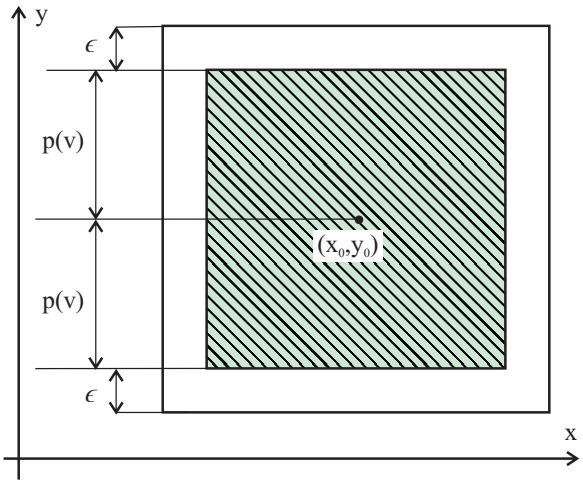

Fig. 24. DynWin safety margin on the robot's plane

7.3.2 *Final step: end-to-end verification for DynWin*

After numerical analysis, we have therefore expressed and proved sound the compilation of the DynWin source operator down to the lowest representation of the HELIX chain, namely *FHCOL*. Remains to exit HELIX per se and compile down to *VIR*.

Theorem in Listing 33 establishes the generic correction of the compilation of arbitrary *FHCOL* operator down to *VIR*, i.e., of the `gen_IR` compilation function. Our work is not quite done: the code produced is a return-less open control flow graph, and we assume from the start memories satisfying our invariant. To fully close the gap, we need to consider the top-level compilation function that additionally:

- allocate globals, and notably two arrays of memories to read the input and output;
- closes the graph generated by `gen_IR` with a final block returning;
- generates a *main* control-flow-graph calling the main one.

This may seem like a very short distance to close, and in some regards it is indeed. Yet, it hides a considerable length of additional necessary developments.

Theory at the level of mutually recursive graphs. First, we had to develop major generic missing infrastructures on the Vellvm side. Indeed, all previously established results had been

```
Theorem DynWin_correct :
  ∀ (a : Vector.t CarrierA 3)          (* parameter *)
    (x : Vector.t CarrierA dynwin_i)   (* input *)
    (y : Vector.t CarrierA dynwin_o), (* y - output *)
      (* Evaluation of the source program. *)
      dynwin_orig a x = y →
      (* if the input data is within bounds for numerical stability, *)
      input_inConstr a x →
      (* and if compilation of the FHCOL program generated
         by the framework compiles down to VIR, *)
      ∀ s pll, compile_w_main (dynwin_fhcolp DynWin_FHCOL_hard)
                              (a++x) newState ≡ inr (s,pll) →
      (* then the resulting code is observationally pure,
         returning a value within bound with the expected output *)
      ∃ g l m r, semantics_llvm pll ≈ Ret (m,(l,(g, r))) ∨ final_rel_val y r.
```

Listing 38. Top-level correctness theorem

developed intra-thread. Although our inter-thread behavior is completely trivial—entering the main, performing a function call, and returning twice—it completely changes the semantic context. Indeed, the semantic model considered in Theorem in Listing 33, built from the composition of `denote_ocfg` and `interp_cfg`, treats function calls as abstract, uninterpreted external events. It does not see the ambient call stack, nor the context required to resolve mutual recursion. In contrast, the semantics of top-level programs ties this recursive knot. This extra infrastructure has been incorporated to Vellvm during the development of this project.

Initialization. Theorem in Listing 33 assumes initial configurations satisfying the memory invariant. On the *FHCOL* side, initialization is rather straight forwards: it is implemented as a shallow Coq function computing the initial memory as a function of the *FHCOL* globals, input and output vectors, and input data. On the Vellvm side however, initialization is itself a quite generic process implemented itself as an ITree computation. It is responsible notably for allocating and initializing globals and function pointers. Here again, we have contributed to Vellvm the necessary infrastructure to reason about this process of initialization, and establish that the invariant is indeed satisfied after initialization.

End-to-end correctness. We are finally ready to put all the pieces together: establishing the correctness of the compilation of the DynWin program down to *VIR*.

This final statement is slightly complicated by two factors:

- compilation to *FHCOL* is not performed via a shallow Coq function, but via meta-programming, we hence need to explicitly refer to the generated code (`DynWin_FHCOL_hard`),
- and we need to account for numerical approximation, proving that inputs with bounds lead to results within bound rather than a simple equality.

But accounting for these two remarks, it captures the desired result.

The proof of this theorem is not completely trivial. It essentially reduces to chaining the following schematic steps:

- Rewriting the source evaluator by the one of *FHCOL* by correctness of the HELIX toolchain;
- Substituting the FHCOL's evaluator by its ITrees denotation;
- Leveraging the proof of relative soundness of initialization to symbolically process *VIR*'s initialization phase, establishing the memory invariant;
- Symbolically processing on the *VIR* side the initial call to main, and its internal call to the operator's function;
- Leveraging Theorem in Listing 33 to fully process the *FHCOL* side, and the subgraph generated by the compilation of the operator on the *VIR* side;
- Symbolically processing the jump to the final block, and both returns, to close the *VIR* computation. This final step requires yet another bit of meta-theory for Vellvm: well-parenthesizing of function calls, capturing the invariants induced by its calling convention.

We stress that the final theorem is specialized to the program of interest, DynWin in this case. In this sense, we could qualify the approach we offer through this project of framework. Almost all of its pieces are entirely generic and reusable. However, the presence of meta-programming forces some manual interaction to wire the produced code. Furthermore, the application-specific numerical, the initialization phase, and the process of stitching together all the results for establishing this final theorem of correctness are intertwined, and must therefore be patched for each application.

Overall, although proving correct a compilation from a source DSL to a general-purpose language such as LLVM IR should be expected to be a challenge, the process has turned out more complex than we had initially anticipated.

8 Results and discussion

8.1 Evaluation

To test our *FHCOL* to *VIR* compiler during development before formally verifying it, we developed a testing framework and defined several compiler tests. 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 intended 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 *VIR*, represented as an 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 6.1.2.

(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 the 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 25.

```
user@dev:~/helix$ make test
Compiling ml/_build/default/testeval.exe
(cd ml; dune build --profile=dev
testeval.exe) 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
```

Fig. 25. A screenshot of the compiler test suite execution

We've defined twelve tests with different levels of complexity, that cover all *FHCOL* operators. 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 E.1.

8.2 Implementation

The current size of HELIX development is 80 KLOC of Coq. The proof-to-specifications ratio is approximately 3:1. There is an insignificant amount of OCaml code (less than 500 lines) implementing the test harness.

The distribution of code in the lines of code (loc) between modules is shown in Table 4. 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

Module	loc	Percent
HCOL	1,160	1.45%
Σ-*HCOL*	10,902	13.62%
MSHCOL	9,023	11.28%
DHCOL	7,218	9.02%
FHCOL	593	0.74%
RHCOL	7,977	9.97%
LLVM	33,985	42.47%
Tactics	93	0.12%
Util	5,676	7.09%
Dynamic Window Monitor	3,390	4.24%
TOTAL	80,017	100.00%

Table 4. Implementation code size

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. The Table 5 shows the percentage of operators used within the DynWin example for each of the HELIX's intermediate languages.

Language	Operators in the DynWin example	Operators in the language	Percent
HCOL	9	18	50%
Σ-*HCOL*	11	14	79%
MSHCOL	9	9	100%
RHCOL	9	10	90%
FHCOL	9	10	90%

Table 5. Percentage of operators covered by the DynWin example

HELIX depends on several 3rd-party tools and libraries such as *ExtLib*, *MathClasses*, *CoLoR*, *Vellvm*, *MetaCoq*, and *Gappa*. It uses *dune* as a build system. As of this writing, a full compilation of HELIX on a modern laptop (13th Generation Intel Core i7-1365U vPro CPU, quad-core, 32GB RAM, SSD) takes about 1 hour. We anticipate that this could be optimised, and compilation time could be significantly shortened, but this is left for future work.

8.3 Future work

HELIX is a prototype system and as such has some limitations. It could be extended in the 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 developed into a code syntehsis 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 been proven for *dynamic window monitor* program. The numerical analysis it involves is described in Section 5.7.3. This is a generic analysis suitable for many problems, but some problems may benefit from custom numerical analysis implementations. To support that, numerical analysis is HELIX is implemented as a customizable module.

Additionally, we identified several interesting future research directions, discussed below.

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.

Finite-sets proofs. In automated proof scenarios involving the application of per-rule lemmas to Σ-*HCOL*, we encounter the generation of sub-goals for additional pre-conditions, predominantly arithmetic goals related to index bounds and set-theoretic goals concerning the (non)intersection, equality, or inclusion of sparsity patterns, represented as finite sets. While arithmetic goals are typically and easily resolved automatically using tactics like `lia`, the set-theoretic goals, which deal with more complex set operations on sparsity patterns, are presently solved manually. It is pertinent to note that these sparsity patterns are represented using the type `Ensemble (FinNat n)`, signifying a set of finite numbers bounded by $n \in \mathbb{N}$. Given the decidable nature of set membership and the generally small size of n, correlating to vector size, the proof obligations arising from set membership are relatively straightforward. Thus, automating the resolution of these sub-goals is not only feasible but practical. While classic Coq automation techniques using *Ltac* could be utilized for this purpose, an alternative and more innovative approach might involve the use of *computation reflection*, as suggested by [56], offering a potentially more efficient and elegant solution.

8.4 Related work

Term rewriting. Encoding program transformations as rewriting rules using the algebraic properties of the language is a well-established approach in the field (see, for example,

[14, 39, 79]). These systems use functional program rewriting without our sparse vector parallelism abstraction, and none of them are formally verified.

Formal reasoning about SPIRAL.. Another attempt to formally verify SPIRAL is described in [16]. 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. Parts of the work in progress in HELIX design and implementation were reported in [91–95].

Certified compilation of imperative languages. Probably the most influential project in the field of certified compilation research is CompCert [48]. It has been used alone or in conjunction with the Verified Software Toolchain (VST) [3] to verify various applications [7, 45, 59, 97], and has spawned many related projects that extend it in various directions [8, 15, 38, 40, 68, 73].

Verified compilation of functional languages. 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 [1] also apply to our approach. Some of their transformation steps could be related to ours. For example, going from dependently typed Σ-*HCOL* to *MSHCOL*, 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* [46], 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* [63]. There are similarities with our approach, as we also use a *definitional interpreter* [66] written in Gallina, to define 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.

Numerical analysis. In our example we use Gappa [25] for numerical analysis. Similar tools, such as FPTaylor [72] or PRECiSA [60], exist, which are also capable of automated error analysis and produce machine-checked proofs of their results. We chose to use Gappa in HELIX for its tight integration into the Coq ecosystem [12].

LLVM IR.. We rely on Vellvm [89] LLVM IR semantics to implement and prove our IR code generation step. This is not the only project aiming to provide formal IR semantics. Other notable projects include Alive [53], Crellvm [44], and K-LLVM [52]. Vellum is under active development and some recent publications are [5] and [87].

Interaction Trees. ITrees [83] is a comprehensive Coq library that facilitates formal modeling and logical reasoning of interactive computations, which are possibly non-terminating,

by representing them as trees of interactions, thus providing a robust tool for the formal verification of complex, concurrent, and reactive systems. Besides Vellvm, there are several other existing applications for it [29, 51, 67, 69, 70, 75, 85], and it is an area of active research [20, 70, 71, 88, 90].

Formal verification of cyber-physical control systems. There is some existing work on formal end-to-end verification of cyber-physical systems. The work closest to ours is VeriPhy [11]. It uses a combination of KeYmaera [37] for *differential dynamic logic* model safety verification, Isabelle/HOL for KeYmaera proof-term checking and arithmetic soundness proofs, and CakeML [46] for verified compilation. From an application point of view, the main difference between HELIX and VeriPhy is that the former is oriented towards high-performance code generation, with much of its pipeline designed to verify code optimisations in intermediate languages. Another difference is that HELIX-generated code uses IEEE floating-point arithmetic, whereas VeriPhy uses fixed-precision integers. Other notable related work includes [26] (*model checking* and *signal temporal logic*), ROSCoq [2] (*logic of events*, *constructive real analysis*, Coq), and VeriDrone [58] (*discrete-time linear temporal logic*, SMT, Coq).

8.5 Contributions

In this section, we delve into the significant contributions and the valuable lessons learned from the HELIX project. This project has made substantial strides in several domains, notably:

Formal methods. Our work has been pivotal in formalizing a class of Operator Languages. We have advanced the field by developing methods to algebraically reason about partial computations using sparse vectors and have introduced a dual semantics approach to build a certified compiler. Additionally, we have explored several methodologies for handling the inherent imprecision of floating point representation, enhancing the robustness and versatility of our computational models.

System development. A notable achievement is the development of an end-to-end system prototype. This prototype is notable for its integration of multiple deep and shallow embedded languages, used in translation validation and certified compilation. Another key accomplishment is the creation of an LLVM-based verified code generation backend for SPIRAL, which is more suitable for more nuanced code generation and formal verification than the existing C-language backend. Furthermore, have demonstrated a formal verification of a mature and complex existing system as opposed to designing a system for verification from the ground up.

Coq developments. In the realm of Coq, our work in monadic sparsity has contributed to the development of more efficient computational models. We have successfully implemented a mechanism to switch between deep and shallow embedding, which offers flexibility in programming and verification. Additionally, we have automated the process of translation validation, simplifying the verification process. Lastly, the use of typeclasses to formalize the properties of operators demonstrates a useful formalization and verification technique.

Vellvm. HELIX is the first front end for Vellvm. The proof of correctness, described in Section 6 has both contributed heavily to the development of Vellvm's meta-theory and opened avenues for future work. As hinted in Lesson 3, the sheer scale of the generated

proofs has led us to make all denotation functions in Vellvm opaque and introduce a symbolic execution mechanism based on rewriting. Reasoning about the lack of failure in the source program led to the notion of the image of a monadic computation, as described in Lesson 2. Finally, the overhead induced by the need for reasoning about the freshness monad used in the compiler highlights the need for a rich DSL of graph combinators for Vellvm, a perspective we touched in Lesson 1. Naturally, at a smaller scale, this project has contributed to enriching the meta-theory of Vellvm as a whole.

Throughout the development of HELIX, we encountered and overcame many challenges. A key insight that we garnered is that formal verification, despite the availability of core tools and techniques, remains a complex and labour-intensive process. This observation underscores the need for further advancements in this field to streamline and simplify the verification process.

8.6 Conclusion

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 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.

We believe 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 the automatic generation of proofs of correctness of the generated code. This is the approach HELIX demonstrates.

APPENDIX

A *HCOL* language

The *HCOL* language comprises operators, each of which is essentially a function that maps from $\mathcal{R}^n$ to $\mathcal{R}^m$. Certain *HCOL* operators necessitate additional parameters, which are required at compile-time. These operators can accept values of type $\mathcal{R}$, as well as functions on $\mathcal{R}$ and $\mathbb{N}$ as compile-time parameters. Currently, HELIX supports two $\mathcal{R}$ constants: `Zero` and `One`, but support for more $\mathcal{R}$ values may be added in the future. HELIX also provides a set of primary binary and unary functions on $\mathcal{R}$, and it offers the ability to construct new functions through λ-abstraction and function application. Furthermore, some *HCOL* operators are higher-order, meaning they accept other operators as parameters, enabling the creation of more complex operators. In some *HCOL* operators, the input has a type in the form $\mathcal{R}^{n+m}$, which can be viewed as functions operating on two distinct arguments, $\mathcal{R}^n$ and $\mathcal{R}^m$. These are passed as one concatenated vector.

A.1 HCOL *syntax*

HCOL syntax is a subset of Coq's syntax and is made up of the following syntactic categories. ⟨Var⟩ represents names for variables that can be introduced by the λ-abstraction. ⟨NatConst⟩ and ⟨CTypeConst⟩ represent literals for values of types $\mathbb{N}$ and $\mathcal{R}$ respectively, while ⟨NTypeExpr⟩ and ⟨CTypeExpr⟩ represent syntax for expressions which evaluate to those types. ⟨CTypeUnFun⟩ and ⟨CTypeBinFun⟩ represent unary and binary functions on $\mathcal{R}$. ⟨CTypeIUnFun⟩ and ⟨CTypeIBinFun⟩ are similar to those but also take additional argument of type $\mathbb{N}$ which represents an index in the input vector. ⟨NatBinFun⟩ represents binary functions on $\mathbb{N}$. ⟨VectorConst⟩ represents $\mathcal{R}^n$ constants. Finally, ⟨HOperator⟩ represents *HCOL* operators.

⟨Var⟩	::=	*variable names*
⟨NatConst⟩	::=	*literals for natural numbers*
⟨CTypeConst⟩	::=	`Zero` | `One`
⟨VectorConstBase⟩	::=	⟨CTypeConst⟩ | ⟨CTypeConst⟩`,` ⟨VectorConstBase⟩
⟨VectorConst⟩	::=	`[`⟨VectorConstBase⟩`]`
⟨CTypeExpr⟩	::=	⟨CTypeConst⟩ | ⟨Var⟩ |
		⟨CTypeUnFun⟩ ⟨CTypeExpr⟩ |
		⟨CTypeBinFun⟩ ⟨CTypeExpr⟩ ⟨CTypeExpr⟩ |
		⟨CTypeIUnFun⟩ ⟨NatExpr⟩ ⟨CTypeExpr⟩ |
		⟨CTypeIBinFun⟩ ⟨NatExpr⟩ ⟨CTypeExpr⟩ ⟨CTypeExpr⟩ |
		`Vnth` ⟨VectorConst⟩ ⟨NatExpr⟩
⟨CTypeUnFun⟩	::=	`abs` | `fun` ⟨Var⟩ => ⟨CTypeExpr⟩
⟨CTypeBinFun⟩	::=	`plus` | `sub` | `mul` | `min` | `max` | `lt` |
		`fun` ⟨Var⟩ ⟨Var⟩ => ⟨CTypeExpr⟩
⟨CTypeIUnFun⟩	::=	`fun` ⟨Var⟩ ⟨Var⟩ => ⟨CTypeExpr⟩

```
⟨CTypeIBinFun⟩  ::=  fun ⟨Var⟩ ⟨Var⟩ ⟨Var⟩ => ⟨CTypeExpr⟩
    ⟨NatExpr⟩   ::=  ⟨NatConst⟩ | ⟨Var⟩ | ⟨NatBinFun⟩ ⟨NatExpr⟩ ⟨NatExpr⟩
  ⟨NatBinFun⟩   ::=  add | sub | mul | fun ⟨Var⟩ ⟨Var⟩ => ⟨NatExpr⟩
  ⟨HOperator⟩   ::=  HPointwise ⟨CTypeIUnFun⟩ | HAtomic ⟨CTypeUnFun⟩ |
                     HScalarProd | HBinOp ⟨CTypeIBinFun⟩ |
                     HReduction ⟨CTypeBinFun⟩ ⟨CTypeExpr⟩ |
                     HEvalPolynomial ⟨VectorConst⟩ |
                     HAppend ⟨VectorConst⟩ | HPrepend ⟨VectorConst⟩ |
                     HMonomialEnumerator ⟨NatExpr⟩ |
                     HInductor ⟨NatExpr⟩ ⟨CTypeBinFun⟩ ⟨CTypeExpr⟩ |
                     HInduction ⟨NatExpr⟩ ⟨CTypeBinFun⟩ ⟨CTypeExpr⟩ |
                     HInfinityNorm | HChebyshevDistance ⟨NatExpr⟩ |
                     HVMinus | HTLess ⟨HOperator⟩ ⟨HOperator⟩ |
                     HCross ⟨HOperator⟩ ⟨HOperator⟩ |
                     HStack ⟨HOperator⟩ ⟨HOperator⟩ |
                     HCompose ⟨HOperator⟩ ⟨HOperator⟩ |
```

A.2 HCOL *operators*

Our current formalization of *HCOL* includes the following operators, shallow-embedded in Coq. In the signatures below, all arguments before semicolon are compile-time parameters.

`HPointwise` $(n\colon \mathbb{N})\,(f\colon \mathbb{I}_n \to \mathcal{R} \to \mathcal{R})\colon \mathcal{R}^n \to \mathcal{R}^n$. This operator applies a function f to each element of the input vector, as shown in Figure 26. The function f takes two arguments: the element's index and its value. The output is the vector of the same length as the input vector.

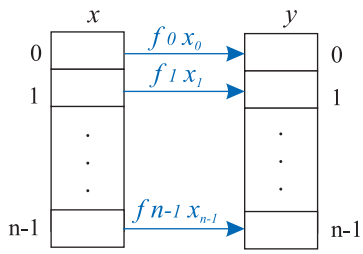

Fig. 26. `HPointwise` operator

`HAtomic` $(f\colon \mathcal{R} \to \mathcal{R})\colon \mathcal{R}^1 \to \mathcal{R}^1$. This operator "lifts" a scalar valued function f to an *HCOL* operator on single element vectors.

`HScalarProd` $(n\colon \mathbb{N})\colon \mathcal{R}^{n+n} \to \mathcal{R}^1$. Calculates the dot product of two vectors. The input vectors must be concatenated and passed as a single vector of size $n + n$. The result is

returned as a single-element vector. For an input vector $\vec{x} = [x_0, x_1, \ldots, x_{n+n-1}]$ it computes $[x_0 \cdot x_n + x_1 \cdot x_{n+1} + \cdots + x_{n-1} \cdot x_{n+n-1}]$.

`HBinOp` $(n\colon \mathbb{N})\ (f\colon \mathbb{I}_n \to \mathcal{R} \to \mathcal{R} \to \mathcal{R})\colon\ \mathcal{R}^{n+n} \to \mathcal{R}^n$. This operator applies a function f to pairs of elements from two halves of an input vector, as shown in Figure 27. The function f additionally takes an index in the range of 0 to $n-1$ as an argument.

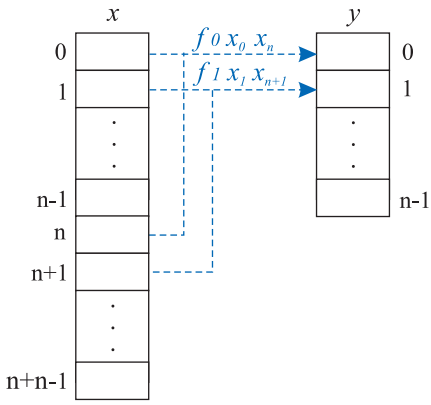

Fig. 27. `HBinOp` operator

`HReduction` $(n\colon \mathbb{N})\ (f\colon \mathcal{R} \to \mathcal{R} \to \mathcal{R})\ (z\colon \mathcal{R})\colon\ \mathcal{R}^n \to \mathcal{R}^1$. This operator performs a *right fold* of a vector. The two parameters are a binary function f and the initial value z. The result is returned as a vector size 1.

`HEvalPolynomial` $(n\colon \mathbb{N})\ (\vec{a}\colon \mathcal{R}^n)\colon\ \mathcal{R}^1 \to \mathcal{R}^1$. This operator computes a univariate polynomial of an n-th degree. It is parameterized by a vector $\vec{a}$ of constant coefficients. The input and output scalar values are represented as vectors of size 1. For input $\vec{x} = [x_0]$ and parameter $\vec{a} = [a_0, a_1, \ldots, a_n]$, it computes $[a_0 + a_1 \cdot x_0 + a_2 \cdot x_0^2 + \cdots + a_n \cdot x_0^n]$.

`HPrepend` $(m\,n\colon \mathbb{N})\ (\vec{a}\colon \mathcal{R}^m)\colon\ \mathcal{R}^n \to \mathcal{R}^{m+n}$. This operator concatenates the input vector of size n with the constant vector "prefix" $\vec{a}$ of size m.

`HAppend` $(m\,n\colon \mathbb{N})\ (\vec{a}\colon \mathcal{R}^m)\colon\ \mathcal{R}^n \to \mathcal{R}^{n+m}$. This operator concatenates the input vector of size n with the constant vector "postfix" $\vec{a}$ of size m.

`HMonomialEnumerator` $(n\colon \mathbb{N})\colon\ \mathcal{R}^1 \to \mathcal{R}^{n+1}$. 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 $[1, x_0, x_0^2, \ldots, x_0^{n-1}]$.

`HInductor` $(n\colon \mathbb{N})\ (f\colon \mathcal{R} \to \mathcal{R} \to \mathcal{R})\ (z\colon \mathcal{R})\colon\ \mathcal{R}^1 \to \mathcal{R}^1$. This operator is a *recursor* [61] 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* [81].

`HInduction` $(n\colon \mathbb{N})\ (f\colon \mathcal{R} \to \mathcal{R} \to \mathcal{R})\ (z\colon \mathcal{R})\colon\ \mathcal{R}^1 \to \mathcal{R}^n$. 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 x0, f (f z x0) x0]`. It is analogous to `FoldList` operator in *Wolfram Mathematica*.

`HInfinityNorm` $(n\colon \mathbb{N})\colon\ \mathcal{R}^n \to \mathcal{R}^1$. This operator computes an *infinity norm* $||\vec{x}||_\infty$ of a given input vector and returns it as a single-element vector.

`HChebyshevDistance` $(n\colon \mathbb{N})\colon\ \mathcal{R}^{n+n} \to \mathcal{R}^1$. 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` $(n\colon \mathbb{N})\colon\ \mathcal{R}^{n+n} \to \mathcal{R}^n$. This operator represents subtraction of two vectors of size n. The vectors are concatenated and passed as an input vector of size $n + n$.

`HCross` $(n_1\ n_2\ m_1\ m_2\colon \mathbb{N})\ (f\colon \mathcal{R}^{m_1} \to \mathcal{R}^{n_1})(g\colon \mathcal{R}^{m_2} \to \mathcal{R}^{n_2})\colon\ \mathcal{R}^{m_1+m_2} \to \mathcal{R}^{n_1+n_2}$. 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 $(\vec{x_0}, \vec{x_1})$ (concatenated) and returns $(f\ \vec{x_0},\ g\ \vec{x_1})$ (also concatenated).

`HStack` $(n_1\ n_2\ m\colon \mathbb{N})\ (f\colon \mathcal{R}^m \to \mathcal{R}^{n_1})\ (g\colon \mathcal{R}^m \to \mathcal{R}^{n_2})\colon\ \mathcal{R}^m \to \mathcal{R}^{n_1+n_2}$. 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 $\vec{x}$ and returns $(f\ \vec{x},\ g\ \vec{x})$ (concatenated).

`HCompose` $(m\ n\ t\colon \mathbb{N})\ (f\colon \mathcal{R}^t \to \mathcal{R}^n)\ (g\colon \mathcal{R}^m \to \mathcal{R}^t)\colon\ \mathcal{R}^m \to \mathcal{R}^n$. This is a higher-order operator for operator composition. When applied to operators f and g it produces a new operator $(f \circ g)$.

`HTLess` $(n\ m_1\ m_2\colon \mathbb{N})\ (f\colon \mathcal{R}^{m_1} \to \mathcal{R}^n)\ (g\colon \mathcal{R}^{m_2} \to \mathcal{R}^n)\colon\ \mathcal{R}^{m_1+m_2} \to \mathcal{R}^n$. 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 $(\vec{x_0}, \vec{x_1})$ (concatenated) and returns a vector produced by element-wise "less than" comparison of $f\ \vec{x_0}$ and $g\ \vec{x_1}$ 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 $\mathcal{R}$.

B Σ-*HCOL* language

Similarly to *HCOL*, Σ-*HCOL* language comprises operators, each of which is essentially a function that maps from $\mathcal{R}^n$ to $\mathcal{R}^m$. However, unlike *HCOL* operators, Σ-*HCOL* operators operate on sparse vectors rather then dense vectors. Σ-*HCOL* is mixed-embedded in Coq: Σ-*HCOL* operators are represented with `SHOperator` record, which contains a function alongside with its sparsity contract. See Section 3.1 for more details.

B.1 Σ-HCOL *syntax*

Σ-*HCOL* syntax is a subset of Coq's syntax and is made up of the following syntactic categories. ⟨Var⟩ represents names for variables that can be introduced by the λ-abstraction.

⟨NatConst⟩ and ⟨CTypeConst⟩ represent literals for values of types $\mathbb{N}$ and $\mathcal{R}$ respectively, while ⟨NTypeExpr⟩ and ⟨CTypeExpr⟩ represent syntax for expressions which evaluate to those types. ⟨CTypeUnFun⟩ and ⟨CTypeBinFun⟩ represent unary and binary functions on $\mathcal{R}$. ⟨CTypeIUnFun⟩ and ⟨CTypeIBinFun⟩ are similar to those but also take an additional argument of type $\mathbb{N}$ which represents an index in the input vector. ⟨NatBinFun⟩ represents binary functions on $\mathbb{N}$. ⟨VectorConst⟩ represents $\mathcal{R}^n$ constants. ⟨CoqProof⟩ represents Coq proof terms. ⟨IndexMap⟩ represents index-mapping functions. ⟨HOperator⟩ and ⟨SHOperator⟩ represent *HCOL* and Σ-*HCOL* operators. Finally, ⟨SHOperatorFamily⟩ represents families of Σ-*HCOL* operators.

⟨Var⟩ ::= *variable names*

⟨NatConst⟩ ::= *literals for natural numbers*

⟨CTypeConst⟩ ::= `Zero` | `One`

⟨VectorConstBase⟩ ::= ⟨CTypeConst⟩ | ⟨CTypeConst⟩`,` ⟨VectorConstBase⟩

⟨VectorConst⟩ ::= `[`⟨VectorConstBase⟩`]`

⟨CoqProof⟩ ::= *Coq proof terms*

⟨HOperator⟩ ::= *HCOL operators, see Appendix A.1*

⟨CTypeExpr⟩ ::= ⟨CTypeConst⟩ | ⟨Var⟩ |
⟨CTypeUnFun⟩ ⟨CTypeExpr⟩ |
⟨CTypeBinFun⟩ ⟨CTypeExpr⟩ ⟨CTypeExpr⟩ |
⟨CTypeIUnFun⟩ ⟨NatExpr⟩ ⟨CTypeExpr⟩ |
⟨CTypeIBinFun⟩ ⟨NatExpr⟩ ⟨CTypeExpr⟩ ⟨CTypeExpr⟩ |
`Vnth` ⟨VectorConst⟩ ⟨NatExpr⟩

⟨CTypeUnFun⟩ ::= `abs` | `fun` ⟨Var⟩ => ⟨CTypeExpr⟩

⟨CTypeBinFun⟩ ::= `plus` | `sub` | `mul` | `min` | `max` | `lt` |
`fun` ⟨Var⟩ ⟨Var⟩ => ⟨CTypeExpr⟩

⟨CTypeIUnFun⟩ ::= `fun` ⟨Var⟩ ⟨Var⟩ => ⟨CTypeExpr⟩

⟨CTypeIBinFun⟩ ::= `fun` ⟨Var⟩ ⟨Var⟩ ⟨Var⟩ => ⟨CTypeExpr⟩

⟨NatExpr⟩ ::= ⟨NatConst⟩ | ⟨Var⟩ | ⟨NatBinFun⟩ ⟨NatExpr⟩ ⟨NatExpr⟩

⟨NatBinFun⟩ ::= `add` | `sub` | `mul` | `fun` ⟨Var⟩ ⟨Var⟩ => ⟨NatExpr⟩

⟨IndexMap⟩ ::= `IndexMap` ⟨NatBinFun⟩ ⟨CoqProof⟩

⟨SHOperator⟩ ::= `Embed` ⟨CoqProof⟩ | `Pick` ⟨CoqProof⟩ |
`Scatter` ⟨IndexMap⟩ | `Gather` ⟨IndexMap⟩ |
`liftM_HOperator` ⟨HOperator⟩ |
`SHPointwise` ⟨CTypeIUnFun⟩ | `SHBinOp` ⟨CTypeIUnFun⟩ |
`SHInductor` ⟨NatExpr⟩ ⟨CTypeBinFun⟩ ⟨CTypeExpr⟩ |
`Apply2Union` ⟨CTypeBinFun⟩ ⟨SHOperator⟩ ⟨SHOperator⟩ |
`SafeCast` ⟨SHOperator⟩ | `UnSafeCast` ⟨SHOperator⟩ |

```
                    SHCompose ⟨SHOperator⟩ ⟨SHOperator⟩ |
                    IReduction ⟨CTypeBinFun⟩ ⟨SHOperatorFamily⟩ |
                    IUnion ⟨CTypeBinFun⟩ ⟨SHOperatorFamily⟩
⟨SHOperatorFamily⟩ ::= fun ⟨Var⟩ => ⟨SHOperator⟩
```

B.2 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` $(s\colon \mathcal{R})\ (n\ b\colon \mathbb{N})\ (bc\colon b < n)$: `SHOperator` $fm\ 1\ n\ s$. Takes an element from a single-element input vector and puts it at a specific index b in a sparse vector of given length.

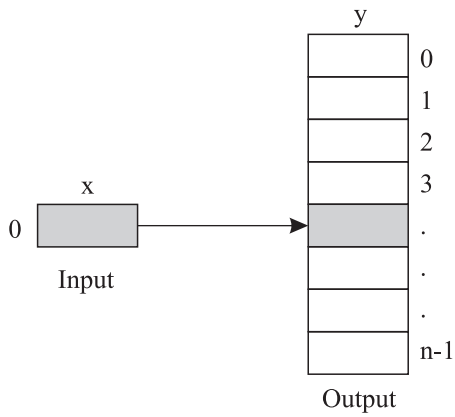

Fig. 28. `Embed` dataflow

`Pick` $(s\colon \mathcal{R})\ (n\ b\colon \mathbb{N})\ (bc\colon b < n)$: `SHOperator` $fm\ n\ 1\ s$. Selects an element from the input vector at the given index b and returns it as a single element vector.

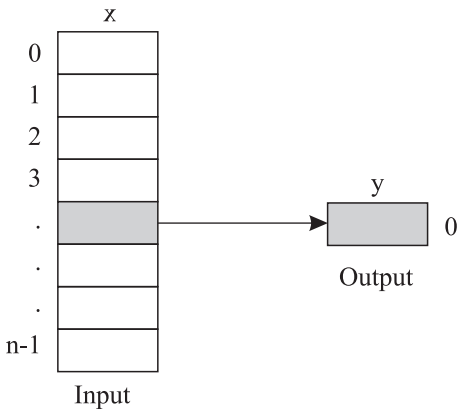

Fig. 29. `Pick` dataflow

`Scatter` $(s : \mathcal{R})$ $(n\ m : \mathbb{N})$ $(f :$ `index_map` $n\ m) :$ `SHOperator` $f m\ n\ m\ s$. 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 needs to be *injective* but not necessarily *surjective*. That means the output vector could be sparse.

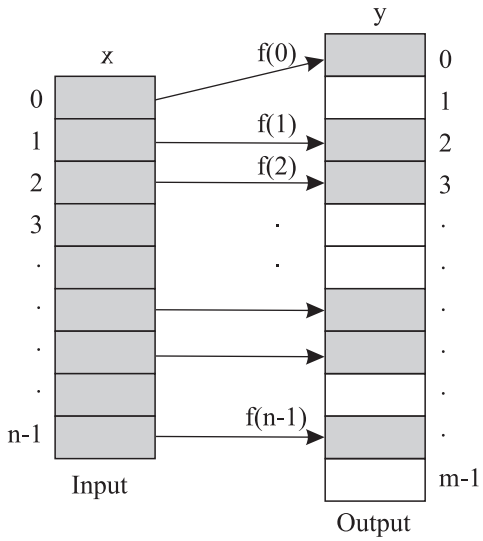

Fig. 30. `Scatter` dataflow

`Gather` $(s : \mathcal{R})$ $(n\ m : \mathbb{N})$ $(f :$ `index_map` $m\ n) :$ `SHOperator` $f m\ n\ m\ s$. 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* f.

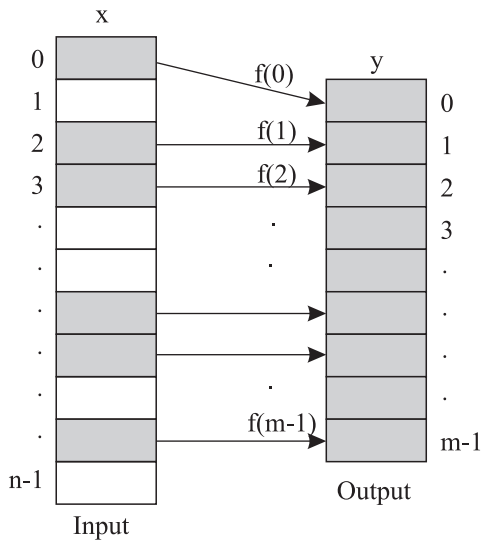

Fig. 31. `Gather` dataflow

`liftM_HOperator` $(i\ o\colon \mathbb{N})$ $(s\colon \mathcal{R})$ $(op\colon$ `avecotr` $i \to$ `avector` $o)\colon$ `SHOperator` $fm\ i\ o\ s$. 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 s. 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` $(s\colon \mathcal{R})$ $(n\colon \mathbb{N})$ $(f\colon \mathbb{I}_n \to \mathcal{R} \to \mathcal{R})\colon$ `SHOperator` $fm\ n\ n\ s$. This is a Σ-*HCOL* version of the `HPointwise` operator from *HCOL* (see Section A.2). 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` $(s\colon \mathcal{R})$ $(n\colon \mathbb{N})$ $(f\colon \mathbb{I}_n \to \mathcal{R} \to \mathcal{R} \to \mathcal{R})\colon$ `SHOperator` $fm\ (n+n)\ n\ s$. This is a Σ-*HCOL* version of the `HBinOp` operator from *HCOL* (see Section A.2). 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` $(s\colon \mathcal{R})$ $(n\colon \mathbb{N})$ $(f\colon \mathcal{R} \to \mathcal{R} \to \mathcal{R})$ $(z\colon \mathcal{R})\colon$ `SHOperator` $fm\ 1\ 1\ s$. This is a Σ-*HCOL* version of the `HInductor` operator from *HCOL* (see Section A.2). The initial value z is treated as *non-structural*.

`Apply2Union` $(i\ o\colon \mathbb{N})$ $(s\colon \mathcal{R})$ $(dot\colon \mathcal{R} \to \mathcal{R} \to \mathcal{R})$ $(f\ g\colon$ `SHOperator` $fm\ i\ o\ s)\colon$ `SHOperator` $fm\ i\ o\ s$. This is a higher-order operator applying two operators to the same input and combining their results using `Vec2Union` (See Listing 8).

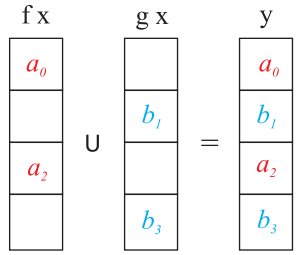

Fig. 32. `Apply2Union` dataflow

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

`SafeCast` (s: $\mathcal{R}$) (i o: $\mathbb{N}$) (f: `SHOperator` `Monoid_`$\mathcal{R}'_{\text{flags}}$ i o s): `SHOperator` `Monoid_`$\mathcal{R}_{\text{flags}}$ i o s. 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 `Monoid_`$\mathcal{R}'_{\text{flags}}$ to `Monoid_`$\mathcal{R}_{\text{flags}}$. As we recall from Section 3.1.1, the former does not track collisions, while the latter does.

`UnSafeCast` (s: $\mathcal{R}$) (i o: $\mathbb{N}$) (f: `SHOperator` `Monoid_`$\mathcal{R}_{\text{flags}}$ i o s): `SHOperator` `Monoid_`$\mathcal{R}'_{\text{flags}}$ i o s. Similar to `SafeCast` but with wrappers switched in the opposite direction, from `Monoid_`$\mathcal{R}_{\text{flags}}$ to `Monoid_`$\mathcal{R}'_{\text{flags}}$.

`SHCompose` (s: $\mathcal{R}$) ($i1$ $o2$ $o3$: $\mathbb{N}$) (g: `SHOperator` $o2$ $o3$ s) (f: `SHOperator` $i2$ $o2$ s): `SHOperator` $i1$ $o3$ s. This performs a functional composition of operators. The f is applied to the input first. The result is then used as an input of g. Sometimes, we use shortcut notation $(g \circledcirc f)$ alluding to the function composition operator $\circ$.

`IReduction` (s: $\mathcal{R}$) (i o n: $\mathbb{N}$) (dot: $\mathcal{R} \to \mathcal{R} \to \mathcal{R}$) ($F$: `SHOperatorFamily` fm i o n s): `SHOperator` i o s. This iteratively applies an indexed family F 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 in Section 3.1.9.

The additional constraint is that the default value for sparse cells (s) is a fixpoint of provided binary function dot, defined as $dot(s, s) = s$. 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` (s: $\mathcal{R}$) (i o n: $\mathbb{N}$) (dot: $\mathcal{R} \to \mathcal{R} \to \mathcal{R}$) ($F$: `SHOperatorFamily` fm i o n s): `SHOperator` i o s. Iteratively applies an indexed family F 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 in Section 3.1.9.

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

This operator is defined for vectors of $\mathcal{R}_\theta$ 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 F members are disjoint. The collision tracking built-in in $\mathcal{R}_\theta$ allows us to prove this.

C *MSHCOL* language

MSHCOL language is similar to Σ-*HCOL*. However, unlike Σ-*HCOL* operators, *MSHCOL* operators operate on memory blocks rather then sparse vectors. *MSHCOL* is also mixed-embedded in Coq: *MSHCOL* operators are represented with `MSHOperator` record, which contains a function on memory blocks alongside with two sets that define input and output memory access patterns. See Section 4.1 for more details.

C.1 MSHCOL *syntax*

MSHCOL syntax is a subset of Coq's syntax and is made up of the following syntactic categories. ⟨Var⟩ represents names for variables that can be introduced by the λ-abstraction. ⟨NatConst⟩ and ⟨CTypeConst⟩ represent literals for values of types $\mathbb{N}$ and $\mathcal{R}$ respectively, while ⟨NTypeExpr⟩ and ⟨CTypeExpr⟩ represent syntax for expressions which evaluate to those types. ⟨CTypeUnFun⟩ and ⟨CTypeBinFun⟩ represent unary and binary functions on $\mathcal{R}$. ⟨CTypeIUnFun⟩ and ⟨CTypeIBinFun⟩ are similar to those but also take additional argument of type $\mathbb{N}$ which represents an index in the input vector. ⟨NatBinFun⟩ represents binary functions on $\mathbb{N}$. ⟨VectorConst⟩ represents $\mathcal{R}^n$ constants. ⟨CoqProof⟩ represents Coq proof terms. ⟨MSHOperator⟩ and ⟨MSHOperatorFamily⟩ represent *MSHCOL* operators and operator families.

⟨Var⟩ ::= *variable names*

⟨NatConst⟩ ::= *literals for natural numbers*

⟨CTypeConst⟩ ::= `Zero` | `One`

⟨VectorConstBase⟩ ::= ⟨CTypeConst⟩ | ⟨CTypeConst⟩ `,` ⟨VectorConstBase⟩

⟨VectorConst⟩ ::= `[`⟨VectorConstBase⟩`]`

⟨CoqProof⟩ ::= *Coq proof terms*

⟨CTypeExpr⟩ ::= ⟨CTypeConst⟩ | ⟨Var⟩ |
⟨CTypeUnFun⟩ ⟨CTypeExpr⟩ |
⟨CTypeBinFun⟩ ⟨CTypeExpr⟩ ⟨CTypeExpr⟩ |
⟨CTypeIUnFun⟩ ⟨NatExpr⟩ ⟨CTypeExpr⟩ |
⟨CTypeIBinFun⟩ ⟨NatExpr⟩ ⟨CTypeExpr⟩ ⟨CTypeExpr⟩ |
`Vnth` ⟨VectorConst⟩ ⟨NatExpr⟩

⟨CTypeUnFun⟩ ::= `abs` | `fun` ⟨Var⟩ => ⟨CTypeExpr⟩

```
⟨CTypeBinFun⟩ ::= plus | sub | mul | min | max | lt |
                  fun ⟨Var⟩ ⟨Var⟩ => ⟨CTypeExpr⟩
⟨CTypeIUnFun⟩ ::= fun ⟨Var⟩ ⟨Var⟩ => ⟨CTypeExpr⟩
⟨CTypeIBinFun⟩ ::= fun ⟨Var⟩ ⟨Var⟩ ⟨Var⟩ => ⟨CTypeExpr⟩
⟨NatExpr⟩ ::= ⟨NatConst⟩ | ⟨Var⟩ | ⟨NatBinFun⟩ ⟨NatExpr⟩ ⟨NatExpr⟩
⟨NatBinFun⟩ ::= add | sub | mul | fun ⟨Var⟩ ⟨Var⟩ => ⟨NatExpr⟩
⟨MSHOperator⟩ ::= MSHEmbed ⟨CoqProof⟩ | MSHPick ⟨CoqProof⟩ |
                  MSHPointwise ⟨CTypeIUnFun⟩ |
                  MSHBinOp ⟨CTypeIUnFun⟩ |
                  MSHInductor ⟨NatExpr⟩ ⟨CTypeBinFun⟩ ⟨CTypeExpr⟩ |
                  MApply2Union ⟨CTypeBinFun⟩
                    ⟨MSHOperator⟩ ⟨MSHOperator⟩
                  MSHCompose ⟨MSHOperator⟩ ⟨MSHOperator⟩ |
                  MSHIReduction ⟨CTypeExpr⟩ ⟨CTypeBinFun⟩
                    ⟨MSHOperatorFamily⟩ |
                  MSHIUnion ⟨MSHOperatorFamily⟩
⟨MSHOperatorFamily⟩ ::= fun ⟨Var⟩ => ⟨MSHOperator⟩
```

C.2 MSHCOL *operators*

Recall from Section 4.1 that there is a 1-to-1 correspondence between *MSHCOL* operators and operators from the final subset of Σ-*HCOL* with the main difference in the input and output container data types: *MSHCOL* uses memory blocks, where Σ-*HCOL* uses sparse vectors.

D *DHCOL* language family

DHCOL is a language family, parametrised by `CType.t` and `NType.t` types which are used to represent data values and integer values respectively. Different instantiations of this parametrisation result in different languages of *DHCOL* family. In HELIX we define two such languages: *RHCOL* and *FHCOL*.

Unlike all the previous languages in HELIX compilation chain, *DHCOL* languages are imperative. *DHCOL* operators can not only compute values, but also modify the memory state. *DHCOL* languages are deep-embedded in Coq: *DHCOL* operators are represented as constructors of the `DSHOperator` inductive type, wich is used to represent an abstract syntax tree of a *DHCOL* program. See Section 5 for more details.

D.1 DHCOL *syntax*

In this section, we present *DHCOL* syntax using BNF notation. As *DHCOL* language family is parametrized by `CType.t` and `NType.t` types, *DHCOL* syntax depends on the syntax for literals used to represent values of those types.

DHCOL syntax is made up of the following syntactic categories. ⟨NatConst⟩, ⟨NTypeConst⟩ and ⟨CTypeConst⟩ represent literals for values of types $\mathbb{N}$, `NType.t` and `CType.t` respectively. ⟨MemBlockConst⟩ represents constant memory blocks. ⟨CTypeUnFun⟩ and ⟨CTypeBinFun⟩ represent unary and binary functions on `CType.t`. ⟨NTypeBinFun⟩ represents binary functions on `NType.t`. ⟨NExpr⟩ and ⟨AExpr⟩ represent syntax for expressions which evaluate to `NType.t` and `CType.t`. ⟨PExpr⟩ represents pointers to memory blocks that can be used to write to memory. ⟨MExpr⟩ represents references to memory blocks that can be used to read from memory. ⟨MemRef⟩ represents pairs of memory references and block sizes. Finally, ⟨DSHOperator⟩ represents *DHCOL* operators.

⟨NatConst⟩ ::= *literasls for natural numbers*
⟨NTypeConst⟩ ::= *literals for NType.t values*
⟨CTypeConst⟩ ::= *literals for CType.t values*
⟨MemBlockConstBase⟩ ::= ⟨NatConst⟩`:` ⟨CTypeConst⟩ |
⟨NatConst⟩`:` ⟨CTypeConst⟩`,` ⟨MemBlockBase⟩
⟨MemBlockConst⟩ ::= {} | {⟨MemBlockConstBase⟩}
⟨NTypeBinFun⟩ ::= `NDiv` | `NMod` | `NPlus` | `NMinus` | `NMult` | `NMin` | `NMax`
⟨CTypeUnFun⟩ ::= `AAbs`
⟨CTypeBinFun⟩ ::= `APlus` | `AMinus` | `AMult` | `AMin` | `AMax` | `AZless`
⟨NExpr⟩ ::= `NVar` ⟨Nat⟩ | `NConst` ⟨NConst⟩ |
⟨NBinFun⟩ ⟨NExpr⟩ ⟨NExpr⟩
⟨PExpr⟩ ::= `PVar` ⟨NatConst⟩
⟨MExpr⟩ ::= `MPtrDeref` ⟨PExpr⟩ |
`MConst` ⟨MemBlockConst⟩ ⟨NTypeConst⟩
⟨AExpr⟩ ::= `AVar` ⟨NatConst⟩ | `AConst` ⟨CTypeConst⟩ |
`ANth` ⟨MExpr⟩ ⟨NExpr⟩ |
⟨CTypeUnFun⟩ ⟨AExpr⟩ |
⟨CTypeBinFun⟩ ⟨AExpr⟩ ⟨AExpr⟩
⟨MemRef⟩ ::= `(`⟨PExpr⟩`,` ⟨NExpr⟩`)`
⟨DSHOperator⟩ ::= `DSHNop` |
`DSHAssign` ⟨MemRef⟩ ⟨MemRef⟩ |
`DSHIMap` ⟨NatConst⟩ ⟨PExpr⟩ ⟨PExpr⟩ ⟨AExpr⟩ |
`DSHBinOp` ⟨NatConst⟩ ⟨PExpr⟩ ⟨PExpr⟩ ⟨AExpr⟩ |
`DSHMemMap2` ⟨NatConst⟩
⟨PExpr⟩ ⟨PExpr⟩ ⟨PExpr⟩ ⟨AExpr⟩ |
`DSHPower` ⟨NExpr⟩ ⟨MemRef⟩ ⟨MemRef⟩
⟨AExpr⟩ ⟨CTypeConst⟩ |
`DSHLoop` ⟨NatConst⟩ ⟨DSHOperator⟩ |

4948 `DSHAlloc` ⟨NTypeConst⟩ ⟨DSHOperator⟩ |
4949 `DSHMemInit` ⟨PExpr⟩ ⟨CTypeConst⟩ |
4950 `DSHSeq` ⟨DSHOperator⟩ ⟨DSHOperator⟩
4951
4952
4953
4954

D.2 DHCOL *operators*

4955 In this section, we present detailed description of all *DHCOL* operators. *DHCOL* operators
4956 are deeply embedded in Coq: they are represented as constructors for the 'DSHOperator'
4957 inductive type.
4958

4959 *`DSHNop`*. This is a "no-op" operator. It does nothing.
4960

4961 *`DSHAssign (src dst: MemRef)`*. An assignment operator copies a single `CType.t` value
4962 from memory location *src* to memory location *dst*.
4963

4964 *`DSHIMap (n: ℕ) (x y: PExpr) (f: AExpr)`*. It iterates the index from 0 to $n-1$. For each
4965 iteration, the values from `x` at the index offset are added to the evaluation context, and the
4966 expression `f` is evaluated. It could be viewed as calling function f with the two arguments: an
4967 index i of type `NType.t` and a value `x[i]` of type `CType.t`. The result is written to `y[i]`.
4968

4969 *`DSHBinOp (n: ℕ) (x y: PExpr) (f: AExpr)`*. This operator is similar to *DSHIMap*. It also
4970 iterates from 0 to $n-1$. For each iteration, it reads two values: at the index offset and at n
4971 plus the index offset from `x` and adds them to the evaluation context and then evaluates the
4972 expression `f`. It could be viewed as calling the function f with the two arguments: `x[i]` and
4973 `x[n+i]`. The result is written to `y[i]`.
4974

4975 *`DSHMemMap2 (n: ℕ) (x0 x1 y: PExpr) (f: AExpr)`*. This is another iterative map. It also
4976 iterates from 0 to $n-1$. For each iteration, it reads `x0[i]` and `x1[i]` and adds them to the
4977 evaluation context and evaluates the expression `f`. It could be viewed as calling the function
4978 f with the two arguments: `x0[i]` and `x1[i]`. The result is written to `y[i]`.
4979

4980 *`DSHPower (n:NExpr) (src dst: MemRef) (f: AExpr) (initial: CT.t)`*. First, it initial-
4981 izes a memory location pointed by `dst` with `initial` value and proceeds to iterate n times.
4982 On each iteration, values are loaded from `src` and `dst`, the `f` is evaluated, and the result
4983 of the evaluation is stored back in `dst`. In C-like pseudo code, it could be expressed as:
4984 `*dst=initial; while(n--){*dst=f(*src,*dst)}`.
4985
4986

4987 *`DSHLoop (n:ℕ) (body: DSHOperator)`*. This is a simple loop. It evaluates a body n times.
4988 For each iteration, the value of the loop index (starting from 0) is put on the top of the
4989 evaluation context before evaluating the *body*.
4990

4991 *`DSHAlloc (size:NT.t) (body: DSHOperator)`*. Lexically scoped memory allocation. It al-
4992 locates a new memory block of size `size`. The newly allocated block is uninitialized. The
4993 new block's offset in memory is put on the top of the evaluation context and then the `body` is
4994 evaluated. The block is only available while the body is being evaluated.
4995

4996 *`DSHMemInit (y: PExpr) (value: CT.t)`*. Initialize all elements of block `y` with given
4997 `value`.
4998

`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.

D.3 DHCOL *semantics*

In this section, we present Big-Step Operational Semantics for the *DHCOL* language, informally described in Section 5. This semantics is implemented as a fixpoint evaluator, described in Section 5.3. In presenting *DHCOL* semantics, we will use the following conventions:

- lower-case names with subscript $_s$ will range over `string`
- lower-case names with subscript $_{\mathbb{N}}$ will range over natural numbers in $\mathbb{N}$
- lower-case names with subscript $_i$ will range over unsigned fixed-length machine integers in `NType.t`
- lower-case names with subscript $_c$ will range over abstract *carrier type* values in `CTtype.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`

D.3.1 The set of states

The set of states has type evalContext×memory, usually denoted as (σ, m). The `evalContext` is a list of variable types and bindings of type `DSHVal`:

```
DSHVal := DSHnatVal value_i | DSHCTypeVal value_c | DSHPtrVal value_N size_i
```

We use square bracket notation $\sigma[\text{index}_{\mathbb{N}}]$ to reference a variable with the de Bruijn index $\text{index}_{\mathbb{N}}$ in the evaluation context σ. The type of $\sigma[\text{index}_{\mathbb{N}}]$ 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 4.1.1. We also use square bracket notation $m[\text{addr}_{\mathbb{N}}]$ to reference a memory block at address $\text{addr}_{\mathbb{N}}$. Thus, the type of $m[\text{addr}_{\mathbb{N}}]$ is `err mem_block`. It will be evaluated to `inr` value_b 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 CType.t`. It will be evaluated to `inr` value_c in case of successful lookup or `inl _` if the offset is out of range.

D.3.2 Expressions

Expressions with types `PExpr`, `MExpr`, `NExpr`, and `AExpr` are evaluated in a given state (σ, m) to their respective values of type `(ℕ×NT.t)`, `(mem_block×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*

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

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

`MExpr` ***evaluation semantics***

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

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

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

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

`NExpr` ***evaluation semantics***

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\langle a_n, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{NMin } a_n\ b_n, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ NMinErr1$$

$$\frac{\langle a_n, \sigma, m\rangle \Downarrow \text{inr } a_i \qquad \langle b_n, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{NMin } a_n\ b_n, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ NMinErr2$$

$$\frac{\langle a_n, \sigma, m\rangle \Downarrow \text{inr } a_i \qquad \langle b_n, \sigma, m\rangle \Downarrow \text{inr } b_i}{\langle \text{NMax } a_n\ b_n, \sigma, m\rangle \Downarrow \text{inr } \max(a_i, b_i)}\ NMaxOK$$

$$\frac{\langle a_n, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{NMax } a_n\ b_n, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ NMaxErr1$$

$$\frac{\langle a_n, \sigma, m\rangle \Downarrow \text{inr } a_i \qquad \langle b_n, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{NMax } a_n\ b_n, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ NMaxErr2$$

`AExpr` *evaluation semantics*

$$\frac{\sigma[x_{\mathbb{N}}] = \text{inr (DSHCTypeVal } v_c)}{\langle \text{AVar } x_{\mathbb{N}}, \sigma, m\rangle \Downarrow \text{inr } v_c}\ AVarOK$$

$$\frac{\sigma[x_{\mathbb{N}}] = \text{inl msg}_s}{\langle \text{AVar } x_{\mathbb{N}}, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ AVarErr1$$

$$\frac{\sigma[x_{\mathbb{N}}] = \text{inr (DSHnatVal } v_i)}{\langle \text{AVar } x_{\mathbb{N}}, \sigma, m\rangle \Downarrow \text{inl } \textit{"invalid AVar type"}}\ AVarErr2$$

$$\frac{\sigma[x_{\mathbb{N}}] = \text{inr (DSHPtrVal } v_{\mathbb{N}}\ \text{size}_i)}{\langle \text{AVar } x_{\mathbb{N}}, \sigma, m\rangle \Downarrow \text{inl } \textit{"invalid AVar type"}}\ AVarErr3$$

$$\frac{}{\langle \text{AConst } v_c, \sigma, m\rangle \Downarrow \text{inr } v_c}\ AConst$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inr } v_c}{\langle \text{AAbs } a_a, \sigma, m\rangle \Downarrow \text{inr } |v_c|}\ AAbsOK$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{AAbs } a_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ AAbsErr$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inr } a_c \qquad \langle b_a, \sigma, m\rangle \Downarrow \text{inr } b_c}{\langle \text{APlus } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inr } (a_c + b_c)}\ APlusOK$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{APlus } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ APlusErr1$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inr } a_c \qquad \langle b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{APlus } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ APlusErr2$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inr } a_c \qquad \langle b_a, \sigma, m\rangle \Downarrow \text{inr } b_c}{\langle \text{AMinus } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inr } (a_c - b_c)}\ AMinusOK$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{AMinus } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ AMinusErr1$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inr } a_c \qquad \langle b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{AMinus } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ AMinusErr2$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inr } a_c \qquad \langle b_a, \sigma, m\rangle \Downarrow \text{inr } b_c}{\langle \text{AMult } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inr } (a_c \cdot b_c)}\ AMultOK$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{AMult } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ AMultErr1$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inr } a_c \qquad \langle b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{AMult } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ AMultErr2$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inr } a_c \qquad \langle b_a, \sigma, m\rangle \Downarrow \text{inr } b_c}{\langle \text{AMin } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inr } \min(a_c, b_c)}\ AMinOK$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{AMin } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ AMinErr1$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inr } a_c \qquad \langle b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{AMin } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ AMinErr2$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inr } a_c \qquad \langle b_a, \sigma, m\rangle \Downarrow \text{inr } b_c}{\langle \text{AMax } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inr } \max(a_c, b_c)}\ AMaxOK$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{AMax } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ AMaxErr1$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inr } a_c \qquad \langle b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{AMax } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ AMaxErr2$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inr } a_c \qquad \langle b_a, \sigma, m\rangle \Downarrow \text{inr } b_c \qquad a_c < b_c}{\langle \text{AZless } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inr one}}\ AZlessOKL$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inr } a_c \qquad \langle b_a, \sigma, m\rangle \Downarrow \text{inr } b_c \qquad a_c \geq b_c}{\langle \text{AZless } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inr zero}}\ AZlessOKR$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{AZless } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ AZlessErr1$$

$$\frac{\langle a_a, \sigma, m\rangle \Downarrow \text{inr } a_c \qquad \langle b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{AZless } a_a\ b_a, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ AZlessErr2$$

$$\frac{\begin{array}{ll}\langle i_n, \sigma, m\rangle \Downarrow \text{inr } i_i & i_i < \text{size}_i \\ \langle x_m, \sigma, m\rangle \Downarrow \text{inr } (x_b, \text{size}_i) & x_b[i_i] = \text{inr } v_c\end{array}}{\langle \text{ANth } x_m\ i_n, \sigma, m\rangle \Downarrow \text{inr } v_c}\ ANthOK$$

$$\frac{\langle i_n, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{ANth } x_m\ i_n, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ ANthErr1$$

$$\frac{\langle i_n, \sigma, m\rangle \Downarrow \text{inr } i_i \qquad \langle x_m, \sigma, m\rangle \Downarrow \text{inl msg}_s}{\langle \text{ANth } x_m\ i_n, \sigma, m\rangle \Downarrow \text{inl msg}_s}\ ANthErr2$$

$$\frac{\langle i_n, \sigma, m\rangle \Downarrow \text{inr } i_i \qquad \langle x_m, \sigma, m\rangle \Downarrow \text{inr } (x_b, \text{size}_i) \qquad i_i \geq \text{size}_i}{\langle \text{ANth } x_m\ i_n, \sigma, m\rangle \Downarrow \text{inl } \textit{"ANth index out of bounds"}}\ ANthErr3$$

$$\frac{\begin{array}{ll}\langle i_n, \sigma, m\rangle \Downarrow \text{inr } i_i & i_i < \text{size}_i \\ \langle x_m, \sigma, m\rangle \Downarrow \text{inr } (x_b, \text{size}_i) & x_b[i_i] = \text{inl msg}_s\end{array}}{\langle \text{ANth } x_m\ i_n, \sigma, m\rangle \Downarrow \text{inl } \textit{"ANth not in memory"}}\ ANthErr4$$

D.3.3 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 (inl *error_message*) or the updated memory m' returned as (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 $\text{block}_b[0 \ldots \text{offset}_{\mathbb{N}}/\text{value}_c]$ means to initialize memory block elements with offsets in the range $[0 \ldots \text{offset}_n)$ with value value_c.

We use from_nat : $\mathbb{N} \to$ 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: (DSHnatVal n_i) :: σ.

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 = inr y, it should be assumed that there is a corresponding rule for the error case x = inl msg_s with inl *error_message* on the right side of the $\Downarrow$ in the conclusion.

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{a_c = x_b[\text{src}_i] \quad b_c = y_b[\text{dst}_i] \quad \sigma' = (\text{DSHCTypeVal } b_c) :: (\text{DSHCTypeVal } a_c) :: \sigma \quad \langle f_a, \sigma', m\rangle \Downarrow inr\ v_c \quad \langle \text{evalPower } n_i\ x_b\ y_b[\text{dst}_i/v_c]\ \text{src}_i\ \text{dst}_i\ f_a\ \sigma, m\rangle \Downarrow \text{inr } y'_b}{\langle \text{evalPower } (n_i + 1)\ x_b\ y_b\ \text{src}_i\ \text{dst}_i\ f_a\ \sigma, m\rangle \Downarrow \text{inr } y'_b}\ evalPowerSn$$

$$\frac{}{\langle \text{DSHLoop } 0\ \text{body}, \sigma, m\rangle \Downarrow \text{inr } m}\ LoopO$$

$$\frac{n_i = \text{from\_nat } n_{\mathbb{N}} \quad \langle \text{DSHLoop } n_{\mathbb{N}}\ \text{body}, \sigma, m\rangle \Downarrow \text{inr } m' \quad \langle \text{body}, ((\text{DSHnatVal } n_i) :: \sigma), m'\rangle \Downarrow \text{inr } m''}{\langle \text{DSHLoop } (n_{\mathbb{N}} + 1)\ \text{body}, \sigma, m\rangle \Downarrow \text{inr } m''}\ LoopSn$$

$$\frac{\langle f, \sigma, m\rangle \Downarrow \text{inr } m' \qquad \langle g, \sigma, m'\rangle \Downarrow \text{inr } m''}{\langle \text{DSHSeq } f\ g, \sigma, m\rangle \Downarrow \text{inr } m''}\ Seq$$

$$\frac{\langle y_p, \sigma, m\rangle \Downarrow \text{inr } (y_{\mathbb{N}}, \text{ysize}_i) \qquad y_b = \text{inr } m[y_{\mathbb{N}}]}{\langle \text{DSHMemInit } y_p\ v_c, \sigma, m\rangle \Downarrow \text{inr } m[y_{\mathbb{N}}/(y_b[0 \ldots \text{ysize}_i/v_c])]}\ MemInit$$

$$\frac{\begin{array}{l}\exists t_i, m[t_i] = \text{inl msg}_s \\ \langle \text{body}, ((\text{DSHPtrVal } t_i\ \text{size}_i) :: \sigma), m[t_i/\text{mem\_empty}]\rangle \Downarrow \text{inr } m'[t_i/]\end{array}}{\langle \text{DSHAlloc size}_i\ \text{body}, \sigma, m\rangle \Downarrow \text{inr } m'}\ Alloc$$

E Examples of generated code

E.1 Dynamic Window Monitor in LLVM IR

HELIX-generated LLVM IR code for the dynamic window monitor is shown in Listing 39. 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:

  %l95 = getelementptr [1 x double], [1 x double]* %a83, i64 0, i64 0
  ; void instr 49
  store double 0x3FF0000000000000, double* %l95, 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:

  %l96 = getelementptr [1 x double], [1 x double]* %a82, i64 0, i64 0
  %l98 = load double, double* %l96, align 8
  %l97 = load double, double* %l95, align 8
  %l99 = fmul double %l97, %l98
  ; void instr 51
  store double %l99, double* %l95, 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:

  %l91 = icmp ult i64 0, 1
  br i1 %l91, label %IMap_loop40, label %MemMap2_entry35
IMap_loop40:
  %IMap_i84 = phi i64 [0, %IMap_entry39], [%IMap_next_i93, %IMap_lcont37]

  br label %IMapLoopBody38
IMapLoopBody38:

  %l85 = getelementptr [1 x double], [1 x double]* %a83, i64 0, i64 %IMap_i84
  %l87 = load double, double* %l85, align 8
  %l88 = getelementptr [3 x double], [3 x double]* @D, i64 0, i64 %Loop_i71
  %l89 = load double, double* %l88, align 8
  %l90 = fmul double %l87, %l89
```

```
  %l86 = getelementptr [1 x double], [1 x double]* %a66, i64 0, i64 %IMap_i84
  ; void instr 45
  store double %l90, double* %l86, align 8
  br label %IMap_lcont37
IMap_lcont37:

  %IMap_next_i93 = add i64 %IMap_i84, 1
  %l92 = icmp ult i64 %IMap_next_i93, 1
  br i1 %l92, label %IMap_loop40, label %MemMap2_entry35
; --- Operator: DSHMemMap2 1 (PVar 1) (PVar 2) (PVar 2) ...---
MemMap2_entry35:

  %l79 = icmp ult i64 0, 1
  br i1 %l79, label %MemMap2_loop36, label %Loop_lcont32
MemMap2_loop36:
  %MemMap2_i72 = phi i64 [0, %MemMap2_entry35], [%MemMap2_next_i81, %MemMap2_lcont33]

  br label %MemMap2LoopBody34
MemMap2LoopBody34:

  %l73 = getelementptr [1 x double], [1 x double]* %a70, i64 0, i64 %MemMap2_i72
  %l76 = load double, double* %l73, align 8
  %l74 = getelementptr [1 x double], [1 x double]* %a66, i64 0, i64 %MemMap2_i72
  %l77 = load double, double* %l74, align 8
  %l78 = fadd double %l76, %l77
  %l75 = getelementptr [1 x double], [1 x double]* %a66, i64 0, i64 %MemMap2_i72
  ; void instr 40
  store double %l78, double* %l75, align 8
  br label %MemMap2_lcont33
MemMap2_lcont33:

  %MemMap2_next_i81 = add i64 %MemMap2_i72, 1
  %l80 = icmp ult i64 %MemMap2_next_i81, 1
  br i1 %l80, label %MemMap2_loop36, label %Loop_lcont32
Loop_lcont32:

  %Loop_loop_next_i108 = add i64 %Loop_i71, 1
  %l107 = icmp ult i64 %Loop_loop_next_i108, 3
  br i1 %l107, 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]
```

```
5713
5714    br label %BinOpLoopBody13
5715  BinOpLoopBody13:
5716
5717    %l32 = getelementptr [2 x double], [2 x double]* %a29, i64 0, i64 %BinOp_i30
5718    %l35 = load double, double* %l32, align 8
5719    %l31 = add i64 %BinOp_i30, 1
5720    %l33 = getelementptr [2 x double], [2 x double]* %a29, i64 0, i64 %l31
5721    %l36 = load double, double* %l33, align 8
5722    %l37 = fsub double %l35, %l36
5723    %l38 = call double @llvm.fabs.f64(double %l37)
5724    %l34 = getelementptr [1 x double], [1 x double]* %a13, i64 0, i64 %BinOp_i30
5725    ; void instr 14
5726    store double %l38, double* %l34, align 8
5727    br label %BinOp_lcont12
5728  BinOp_lcont12:
5729
5730    %BinOp_next_i41 = add i64 %BinOp_i30, 1
5731    %l40 = icmp ult i64 %BinOp_next_i41, 1
5732    br i1 %l40, label %BinOp_loop15, label %MemMap2_entry10
5733  ; --- Operator: DSHMemMap2 1 (PVar 1) (PVar 2) (PVar 2) ...---
5734  MemMap2_entry10:
5735
5736    %l26 = icmp ult i64 0, 1
5737    br i1 %l26, label %MemMap2_loop11, label %Loop_lcont7
5738  MemMap2_loop11:
5739    %MemMap2_i19 = phi i64 [0, %MemMap2_entry10], [%MemMap2_next_i28, %MemMap2_lcont8]
5740
5741    br label %MemMap2LoopBody9
5742  MemMap2LoopBody9:
5743
5744    %l20 = getelementptr [1 x double], [1 x double]* %a17, i64 0, i64 %MemMap2_i19
5745    %l23 = load double, double* %l20, align 8
5746    %l21 = getelementptr [1 x double], [1 x double]* %a13, i64 0, i64 %MemMap2_i19
5747    %l24 = load double, double* %l21, align 8
5748    %l25 = call double @llvm.maxnum.f64(double %l23, double %l24)
5749    %l22 = getelementptr [1 x double], [1 x double]* %a13, i64 0, i64 %MemMap2_i19
5750    ; void instr 9
5751    store double %l25, double* %l22, align 8
5752    br label %MemMap2_lcont8
5753  MemMap2_lcont8:
5754
5755    %MemMap2_next_i28 = add i64 %MemMap2_i19, 1
5756    %l27 = icmp ult i64 %MemMap2_next_i28, 1
5757    br i1 %l27, label %MemMap2_loop11, label %Loop_lcont7
5758  Loop_lcont7:
5759
5760    %Loop_loop_next_i60 = add i64 %Loop_i18, 1
5761    %l59 = icmp ult i64 %Loop_loop_next_i60, 2
5762
5763
```

```
  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
  %l9 = uitofp i1 %l8 to double
  %l5 = getelementptr [1 x double], [1 x double]* %Y, i64 0, i64 %BinOp_i1
  ; void instr 1
  store double %l9, double* %l5, 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 39. Dynamic Window Monitor in LLVM IR

E.2 Dynamic Window Monitor in C

SPIRAL-generated C code for the dynamic window monitor is shown in listings below. Listing 40 shows the code that is generated for generic x86 target platform without SIMD instructions, while Listing 41 shows the code for an x86 platform with SIMD instructions extension enabled. See Section 7.2 for more details about compilation with SPIRAL.

Listing 40. 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 41. Dynamic Window Monitor in C (optimized)

```
#include <float.h>
#include <smmintrin.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_loaddup_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;
}
```

References

[1] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*.

[2] Abhishek Anand and Ross A. Knepper. 2015. ROSCoq: Robots Powered by Constructive Reals. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9236)*, Christian Urban and Xingyuan Zhang (Eds.). Springer, 34–50. https://doi.org/10.1007/978-3-319-22102-1_3

[3] Andrew W. Appel. 2011. Verified Software Toolchain. In *Programming Languages and Systems*, Gilles Barthe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

[4] Andrew W Appel, Lennart Beringer, Adam Chlipala, Benjamin C Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017), 20160331.

[5] Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, and Steve Zdancewic. 2024. A Two-Phase Infinite/Finite Low-Level Memory Model. *Proc. ACM Program. Lang.* ICFP (aug 2024). https://doi.org/10.1145/3674652

[6] Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. *SIGPLAN Not.* 39, 1 (Jan. 2004), 14–25. https://doi.org/10.1145/982962.964003

[7] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *Proceedings of the 24th USENIX Conference on Security Symposium*

(Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 207–221.

[8] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. 2014. Verified Compilation for Shared-Memory C. In *ESOP (LNCS, Vol. 8410)*. 107–127. https://doi.org/10.1007/978-3-642-54833-8_7

[9] Michael Blair, Sally Obenski, and Paula Bridickas. 1992. Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia.

[10] Frédéric Blanqui and Adam Koprowski. 2011. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science* 21, 4 (2011), 827–859.

[11] Rose Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. 2018. VeriPhy: verified controller executables from verified cyber-physical system models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 617–630. https://doi.org/10.1145/3192366.3192406

[12] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. 2009. Combining Coq and Gappa for certifying floating-point programs. In *International Conference on Intelligent Computer Mathematics*. Springer, 59–74.

[13] Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A unified library for proving floating-point algorithms in Coq. In *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*. IEEE, 243–252.

[14] Peter Borovanskỳ, Claude Kirchner, Hélene Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. 1998. An overview of ELAN. *Electronic Notes in Theoretical Computer Science* 15 (1998), 55–70.

[15] Timothy Bourke, Lélio Brun, and Marc Pouzet. 2020. Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset. In *Proceedings of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages* (New Orleans, LA, USA), Vol. 4. Association for Computing Machinery. https://doi.org/10.1145/3371112

[16] Patrick J. Brinich. 2020. *Formal Verification of Spiral Generated Code*. Master's thesis. Drexel University.

[17] Venanzio Capretta. 2005. General recursion via coinductive types. *Log. Methods Comput. Sci.* 1, 2 (2005). https://doi.org/10.2168/LMCS-1(2:1)2005

[18] Pierre Castéran and Matthieu Sozeau. 2012. *A gentle introduction to type classes and relations in Coq*. Technical Report. Technical Report hal-00702455, version 1.

[19] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proc. ACM Program. Lang.* 7, POPL (2023), 1770–1800. https://doi.org/10.1145/3571254

[20] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages* 7, POPL (2023).

[21] Robert N Charette. 2009. This car runs on code. *IEEE spectrum* 46, 3 (2009), 3.

[22] Adam Chlipala. 2017. *Formal reasoning about programs*. http://adam.chlipala.net/frap

[23] Minki Cho, Youngju Song, Dongjae Lee, Lennard Gäher, and Derek Dreyer. 2023. Stuttering for Free. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1677–1704. https://doi.org/10.1145/3622857

[24] Bruce Dawson. 2012. *Comparing Floating Point Numbers*. https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

[25] Florent De Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. 2008. Certifying floating-point implementations using Gappa. *arXiv preprint arXiv:0801.0523* (2008).

[26] Ankush Desai, Tommaso Dreossi, and Sanjit A. Seshia. 2017. Combining Model Checking and Runtime Verification for Safe Robotics. In *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10548)*, Shuvendu K. Lahiri and Giles Reger (Eds.). Springer, 172–189. https://doi.org/10.1007/978-3-319-67531-2_11

[27] The Coq development team. 2004. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.0.

[28] Christof Ebert and Capers Jones. 2009. Embedded Software: Facts, Figures, and Future. *Computer* 42, 4 (April 2009), 42–52. https://doi.org/10.1109/MC.2009.118

[29] Simon Foster, Chung-Kil Hur, and Jim Woodcock. 2021. Formally Verified Simulations of State-Rich Processes using Interaction Trees in Isabelle/HOL. arXiv:2105.05133 [cs.LO]

[30] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. 1997. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine* 4, 1 (1997), 23–33.

[31] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. 2009. Operator Language: A Program Generation Framework for Fast Kernels. In *IFIP Working Conference on Domain Specific Languages (DSL WC) (Lecture Notes in Computer Science, Vol. 5658)*. Springer, 385–410.

[32] 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. 2017. High-Assurance SPIRAL: End-to-End Guarantees for Robot and Car Control. *IEEE Control Systems* 37, 2 (April 2017), 82–103. https://doi.org/10.1109/MCS.2016.2643244

[33] Franz Franchetti, Tze-Meng Low, Thom Popovici, Richard Veras, Daniele G. Spampinato, Jeremy Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proceedings of the IEEE, special issue on "From High Level Specification to High Performance Code"* 106, 11 (2018).

[34] Franz Franchetti and Markus Püschel. 2009. Generating High-Performance Pruned FFT Implementations. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 549–552.

[35] Franz Franchetti, Markus Püschel, José M. F. Moura, and Christoph W. Ueberhuber. 2002. Short Vector SIMD Code Generation for DSP Algorithms. In *High Performance Extreme Computing (HPEC)*.

[36] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. 2005. Formal Loop Merging for Signal Transforms. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. ACM, New York, NY, USA, 315–326. https://doi.org/10.1145/1065010.1065048

[37] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. 2015. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In *International Conference on Automated Deduction*. Springer, 527–538.

[38] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

[39] Bastian Hagedorn, Johannes Lenfers, Thomas Kundefinedhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving High-Performance the Functional Way: A Functional Pearl on Expressing High-Performance Optimizations as Rewrite Strategies. *Proc. ACM Program. Lang.* 4, ICFP, Article 92 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408974

[40] Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards certified separate compilation for concurrent programs. In *PLDI*. 111–125. https://doi.org/10.1145/3314221.3314595

[41] Michael A Johnson and Mohammad H Moradi. 2005. *PID control*. Springer.

[42] Rudolph Emil Kalman. 1960. A new approach to linear filtering and prediction problems. (1960).

[43] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A Formal C Memory Model Supporting Integer-Pointer Casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 326–335. https://doi.org/10.1145/2737924.2738005

[44] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. 2018. Crellvm: Verified Credible Compilation for LLVM. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 631–645. https://doi.org/10.1145/3192366.3192377

[45] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. ACM, New York, NY, USA, 234–248. https://doi.org/10.1145/3293880.3294106

[46] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. *ACM SIGPLAN Notices* 49, 1 (2014), 179–191.

[47] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[48] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[49] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA. 26 pages. https://hal.inria.fr/hal-00703441

[50] Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. 2022. C4: verified transactional objects. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–31. https://doi.org/10.1145/3527324

[51] Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. 2022. C4: verified transactional objects. *Proc. ACM Program. Lang.* 6, OOPSLA (2022), 1–31. https://doi.org/10.1145/3527324

[52] Liyi Li and Elsa Gunter. 2020. K-LLVM: A Relatively Complete Semantics of LLVM IR. In *34rd European Conference on Object-Oriented Programming, ECOOP 2020, Berlin, Germany*. https://doi.org/10.4230/LIPIcs.ECOOP.2020.7

[53] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

[54] Tze-Meng Low and Franz Franchetti. 2017. High Assurance Code Generation for Cyber-Physical Systems. In *IEEE International Symposium on High Assurance Systems Engineering (HASE)*.

[55] Vicente Macian, Jose Manuel Lujan, Carlos Guardiola, and Pedro Yuste. 2006. DFT-based controller for fuel injection unevenness correction in turbocharged diesel engines. *IEEE transactions on control systems technology* 14, 5 (2006), 819–827.

[56] Gregory Malecha. 2017. Speeding Up Proofs with Computational Reflection. https://gmalecha.github.io/reflections/2017/speeding-up-proofs-with-computational-reflection. Accessed: 2020-09-01.

[57] Gregory Malecha et al. 2012. ExtLib Coq library. https://github.com/coq-ext-lib/coq-ext-lib. Accessed: 2020-08-18.

[58] Gregory Malecha, Daniel Ricketts, Mario M. Alvarez, and Sorin Lerner. 2016. Towards foundational verification of cyber-physical systems. In *2016 Science of Security for Cyber-Physical Systems Workshop, SOSCYPS@CPSWeek 2016, Vienna, Austria, April 11, 2016*. IEEE Computer Society, 1–5. https://doi.org/10.1109/SOSCYPS.2016.7580000

[59] William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A Verified Messaging System. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 87 (oct 2017), 28 pages. https://doi.org/10.1145/3133911

[60] Mariano Moscato, Laura Titolo, Aaron Dutle, and César A Munoz. 2017. Automatic estimation of verified floating-point round-off errors via static analysis. In *Computer Safety, Reliability, and Security: 36th International Conference, SAFECOMP 2017, Trento, Italy, September 13-15, 2017, Proceedings 36*. Springer, 213–229.

[61] Yiannis N Moschovakis. 1989. A mathematical modeling of pure, recursive algorithms. In *International Symposium on Logical Foundations of Computer Science*. Springer, 208–229.

[62] José M. F. Moura, Jeremy Johnson, Robert W. Johnson, David Padua, Viktor K. Prasanna, Markus Püschel, Bryan Singer, Manuela Veloso, and Jianxin Xiong. 2001. Generating Platform-Adapted DSP Libraries using SPIRAL. In *High Performance Extreme Computing (HPEC)*.

[63] Scott Owens, Magnus O Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional big-step semantics. In *European Symposium on Programming*. Springer, 589–615.

[64] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems: 4th International Conference, TACAS'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28–April 4, 1998 Proceedings 4*. Springer, 151–166.

[65] M. Püschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (Feb 2005), 232–275. https://doi.org/10.1109/JPROC.2004.840306

[66] John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. Association for Computing

Machinery, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

[67] Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-Language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL, Article 27 (jan 2023), 31 pages. https://doi.org/10.1145/3571220

[68] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22. https://doi.org/10.1145/2487241.2487248

[69] Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. 2023b. Semantics for Noninterference with Interaction Trees. In *Proceedings of the 37th Annual European Conference on Object-Oriented Programming (ECOOP 2023)*.

[70] Lucas Silver, Eddy Westbrook, Matthew Yacavone, and Ryan Scott. 2023a. Interaction Tree Specifications: A Framework for Specifying Recursive, Effectful Computations That Supports Auto-Active Verification. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:26. https://doi.org/10.4230/LIPIcs.ECOOP.2023.30

[71] Lucas Silver and Steve Zdancewic. 2021. Dijkstra Monads Forever: Termination-Sensitive Specifications for Interaction Trees. *Proc. ACM Program. Lang.* 5, POPL, Article 26 (Jan. 2021), 28 pages. https://doi.org/10.1145/3434307

[72] Alexey Solovyev, Marek S Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 1 (2018), 1–39.

[73] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2019. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371091

[74] Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL (2023), 1121–1151. https://doi.org/10.1145/3571232

[75] Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 39 (jan 2023), 31 pages. https://doi.org/10.1145/3571232

[76] Matthieu Sozeau. 2010. A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning* (2010), 1–12.

[77] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *Journal of Automated Reasoning* (2020), 1–53.

[78] Bas Spitters and Eelis Van der Weegen. 2011. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science* 21, 4 (2011), 795–825.

[79] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code. *SIGPLAN Not.* 50, 9 (Aug. 2015), 205–217. https://doi.org/10.1145/2858949.2784754

[80] The GAP Group 2020. *GAP – Groups, Algorithms, and Programming, Version 4.11.0*. The GAP Group. https://www.gap-system.org

[81] Stephen Wolfram et al. 1999. *The MATHEMATICA® book, version 4*. Cambridge university press.

[82] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction Trees. *Proceedings of the ACM on Programming Languages* 4, POPL (2020). https://doi.org/10.1145/3371119

[83] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction Trees. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020).

[84] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.

[85] Kangfeng Ye, Simon Foster, and Jim Woodcock. 2022. Formally verified animation for RoboChart using Interaction Trees. In *International Conference on Formal Engineering Methods*. Springer, 404–420.

[86] Kangfeng Ye, Simon Foster, and Jim Woodcock. 2023. Formally Verified Animation for RoboChart using Interaction Trees. *CoRR* abs/2303.09106 (2023). https://doi.org/10.48550/ARXIV.2303.09106 arXiv:2303.09106

[87] Euisun Yoon. 2023. *Modular semantics and metatheory for LLVM IR*. Ph. D. Dissertation. University of Pennsylvania.

[88] Irene Yoon, Yannick Zakowski, and Steve Zdancewic. 2022. Formal Reasoning About Layered Monadic Interpreters. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022).

[89] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, Compositional, and Executable Formal Semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP, Article 67 (aug 2021), 30 pages. https://doi.org/10.1145/3473572

[90] Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. 2020. An Equational Theory for Weak Bisimulation via Generalized Parameterized Coinduction. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. https://doi.org/10.1145/3372885.3373813

[91] Vadim Zaliva. 2020. *HELIX: From Math to Verified Code*. Ph. D. Dissertation. Carnegie Mellon University.

[92] Vadim Zaliva and Franz Franchetti. 2015. Formal Verification of HCOL Rewriting. (2015).

[93] Vadim Zaliva and Franz Franchetti. 2018. HELIX: A Case Study of a Formal Verification of High Performance Program Generation. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing* (St. Louis, MO, USA) *(FHPC 2018)*. ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/3264738.3264739

[94] Vadim Zaliva and Matthieu Sozeau. 2019. Reification of Shallow-Embedded DSLs in Coq with Automated Verification. In *International Workshop on Coq for Programming Languages (CoqPL)*.

[95] Vadim Zaliva, Ilia Zaichuk, and Franz Franchetti. 2020. 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)*.

[96] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference) (LIPIcs, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:19. https://doi.org/10.4230/LIPICS.ITP.2021.32

[97] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, William Mansky, Benjamin Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 32:1–32:19. https://doi.org/10.4230/LIPIcs.ITP.2021.32

[98] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2103621.2103709

[99] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2013. Formal Verification of SSA-Based Optimizations for LLVM. In *Proc. 2013 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*. https://doi.org/10.1145/2499370.2462164