Scimitar: Functional Programs as Optimization Problems

Nate F. F. Bragg
Tufts University
Medford, MA, USA
nate@cs.tufts.edu

Jeffrey S. Foster
Tufts University
Medford, MA, USA
jeffrey.foster@tufts.edu

Philip Zucker
Draper Laboratory
Cambridge, MA, USA
pzucker@draper.com

Abstract

Mixed integer linear programming is a powerful and widely used approach to solving optimization problems, but its expressiveness is limited. In this paper we introduce the optimization-aided language Scimitar, which encodes optimization problems using an expressive functional language, with a compiler that targets a mixed integer linear program solver. Scimitar provides easy access to encoding techniques that normally require expert knowledge, enabling solve-time conditional constraints, inlining, loop unrolling, and many other high-level language constructs. We give operational semantics for Scimitar and constraint encodings of various features. To demonstrate Scimitar, we present a number of examples and benchmarks including classic optimization domains and more complex problems. Our results indicate that Scimitar's use of a dedicated MILP solver is effective for expressively modeling optimization problems embedded within functional programs.

CCS Concepts: • **Theory of computation** → **Integer programming**; **Linear programming**; • **Software and its engineering** → **Compilers**; **Functional languages**.

Keywords: mixed integer linear programming, functional programming, compilers

ACM Reference Format:
Nate F. F. Bragg, Jeffrey S. Foster, and Philip Zucker. 2024. Scimitar: Functional Programs as Optimization Problems. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '24), October 23–25, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689492.3690051

1 Introduction

Mixed integer linear programming (*MILP*) is a classic constraint optimization approach for problems with linear and integer variables and constraints. It has applications ranging from simple to complex, including network and logistics problems, and machine learning.

Despite the success of MILP, in practice it can still be very challenging to write MILP programs. The difficulty lies in MILP's limited expressiveness, which means that MILP programmers must use complex encodings to map the semantics of their problems into the MILP language. For example, encodings enable MILP to be applied to domains such as boolean logic [Braun 2012], nonlinear functions like multiplication [Dombrowski 2018] or piecewise linear functions [Huchette and Vielma 2022], conditional constraints [GAMS Development Corporation 2022; Huchette and Vielma 2019; Ruiz et al. 2012], and others [Pardalos 1988; Wikipedia contributors 2024]. However, while such encodings are effective, they require significant expert knowledge to use, and they make programs difficult to maintain as they complicate and obscure the underlying problem.

In this paper we introduce Scimitar, an *optimization-aided language* that enables seamless integration of MILP constraint optimization problems into functional programs. Several researchers have explored integrating constraint solvers into programming languages, e.g., logic languages like SWI-Prolog [Wielemaker et al. 2012], verification-aware languages like Dafny [Leino 2010], prover languages like Why [Filliâtre 2003], and the Rosette solver-aided language [Torlak and Bodík 2013]. However, these languages focus on SMT or similar decision procedures. In contrast, the design and implementation of Scimitar show how to integrate an *optimization solver* into functional programming.

Our approach is inspired by the SMT solver-aided language Rosette. Like Rosette, Scimitar combines functional and symbolic reasoning in the same program, but unlike Rosette, Scimitar is optimization-aided—specifically by a MILP solver—rather than SMT-aided. Scimitar programs include the `(minimize o e)` construct, which minimizes the objective `o` with respect to the constraints in the body expression `e`. The key novelty of Scimitar is that `e` is also a functional expression containing function calls, conditionals, loops, and more, which is compiled by Scimitar into optimization problems. This relieves Scimitar programmers of the burden of manually encoding these constructs, enabling a level of abstraction not available in normal optimization

Onward! '24, October 23–25, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1215-9/24/10
https://doi.org/10.1145/3689492.3690051

problem representations. In Section 2 we give two examples demonstrating our language design while explaining Scimitar's behavior in detail.

Another difference is that Scimitar separates host semantics from the solver semantics used in the body of `minimize`. Solver semantics translate source code into MILP problems of the *standard form* $\operatorname{argmin} c^T x \text{ s.t. } Ax \preceq b$. Scimitar also includes a type system that tracks variable bounds, which are required by optimization solvers. In Section 3 we give a grammar and formal semantics for Scimitar that precisely captures how functional and optimization code interact.

Scimitar uses a wide range of encodings to lower functional programming features to optimization code, while Rosette relies directly on Z3's smtlib2 encoding and built-in translations. Scimitar has encodings for features such as multiplication, booleans, conditionals, and dynamic vector indexing. Since solver problems are finite, Scimitar must also perform function call inlining and loop unrolling during compilation. Section 4 describes these encodings.

Finally, the applications of these languages are quite distinct. Those given by Rosette's developers include formal verification of systems such as JIT compilers, synthesis of GPU kernels, program repair, and model checking. On the other hand, Scimitar targets applications such as logistics problems, resource allocation, and design optimization. In Section 5 we present Scimitar benchmarks in different domains to demonstrate the capabilities in our system's design. We measure performance, giving evidence that a dedicated MILP solver compares favorably with the more general purpose SMT solver on optimization applications.

In summary, the contributions of this paper include:

- Scimitar, a new language that combines functional and symbolic reasoning to provide a set of abstractions and corresponding encodings to seamlessly compose MILP constraint optimization problems into functional programs (Section 2),
- A formal semantics for a functional host language with nested optimization problems solved via a translation into standard form (Section 3),
- An implementation strategy for compiling programs into MILP encodings (Section 4), and
- An evaluation demonstrating that Scimitar programs can efficiently solve problems in a variety of domains (Section 5).

2 Examples

Scimitar is a unique language design that lets a user combine optimization with classical computation seamlessly with a clear phase distinction. It combines these paradigms through an integrated MILP solver, which is controlled via constraint assertions and an optimization construct. Scimitar provides functional features including recursive, anonymous, and (limited) higher-order functions; conditionals; let bindings; and values including numbers, booleans, and tuples.

```
1  (minimize n
2    (letrec
3      ((sum-to-n
4         (lambda (n acc)
5           (if (= n 0.0) acc
6               (sum-to-n (- n 1.0) (+ n acc))))))
7      (assert (>= (sum-to-n n 0.0) 100.0))))
```

Figure 1. Example of a recursive function

In this section, we introduce Scimitar via two examples, a recursive summation function and an arena allocator. Section 5 walks through several more examples.

2.1 Sum from Zero to N

Scimitar programs are written as a specialized language within Racket. In addition to standard functional language features, Scimitar includes a form (`minimize o e`), where expression `o` is the objective to minimize subject to constraints in the expression `e`. For example, Figure 1 shows a simple Scimitar program that finds the minimum value of `n` subject to the constraint $\sum_{i=0}^{n} i \geq 100.0$. This constraint uses the recursive function `sum-to-n`, defined within the minimization expression with two variables `n` and `acc`. On line 5, if `n` is zero, `sum-to-n` returns `acc`. Otherwise it recurses, decrementing `n` and adding `n` to the accumulator. When executed, this program gives a final result of $n = 14$.

Normal program semantics executes with a known n and calculates a final value sequentially. On the other hand, solver semantics must model the whole program at once, divining the initial n from the constraint on the result value.

The recursion in `sum-to-n` effectively creates a constraint against the conditional branches at each recursion depth. Recursion is handled via inlining, which relies on a sentinel indicating dynamically that the inlining depth has been reached. The constraint on line 7 is finally satisfied when $n + (n-1) \ldots + 0.0 \geq 100.0$, and the value of n is chosen.

In solver semantics, the entire **`if`** expression on lines 5-6 is active, including the guard and both branches. Branches taken can be picked dynamically, and can even decide the guard. This example forces the solver to choose between branches according to the accumulator value rather than n.

However, during normal execution both branches cannot be active simultaneously, so the solver must have a way to enable one and disable the other depending on the result of the condition expression. This can not be done directly, but it can be accomplished indirectly using the encoding $[\![\text{if } c \text{ then } t \text{ else } f]\!] = c \cdot t + (1-c) \cdot f$. The condition expression (`= n 0.0`) has a binary type, and we use that as the indicator variable c. If c is true, the result of the true branch t is multiplied by one, and the false f by zero, and vice versa. I.e., if $c = 0$, then $0 \cdot t + (1-0) \cdot f = f$, which selects the false branch. We expand this multiplication using McCormick envelopes, which we discuss below in Section 2.2.

```
1  (define arena (Init-Arena))
2  (define (Allocate size)
3    (let ((bucket-ix (Bucket-For-Size size)))
4      ...
5      (Update-Arena-Dist arena)
6      (Grab-Block bucket-ix)))
```

Figure 2. Allocate example: `Allocate`

Scimitar's *functional language compiler* translates this example to a restricted constraint language via this encoding as well as many others. From there, the *optimization language compiler* outputs a matrix format understood by Scimitar's virtual machine, which repeatedly invokes a MILP solver and decodes the optimum into a continuation to execute.

This example demonstrates how in the high level language users can reason about each constraint sub-problem independently, and Scimitar does the heavy lifting of composing these into an overall problem. By considering a term in isolation, users can ignore its structural relationship with adjacent terms. While this does not make reasoning about the solver's decision process for a particular program any easier, it does provide some guarantees about the program's construction. This is in contrast to normal optimization problems, where users must put in effort to tie parts of their problem together, which later on could represent a maintainability hurdle. Our goal is to allow users to implement certain classes of problems, including traditional optimization problems such as those that we present in Section 5.

2.2 Arena Allocator

The previous section gives a basic example of a top-level `minimize` expression, but in general Scimitar allows such expressions within of a larger program. This enables iteratively solving the problem. Figures 2 and 3 explore Scimitar's *host-solver boundary* with a more complex example.

Here, we demonstrate a memory allocator like malloc. Some malloc implementations use an arena of buckets based on allocation size, with allocations drawn from the least greater sized bucket. When users request a block of memory, the allocator retrieves one from the appropriate bucket. Over time, some buckets are used more than others, and the logical solution is to rebalance the buckets to prevent that.

Figure 2 defines `Allocate`, a stateful allocator function that follows this design. It is responsible for the mutable state, including the current arena as well as the history of arenas, used for hysteresis. On line 1 we initialize the global constant `arena`, the number of buckets for each size. The exact blocks the arena uses are managed outside of the optimization, and are not shown.

Line 3 selects the bucket containing the requested block, which is then updated to ensure one is available. This bucket is used in various other operations that we omit for brevity. The call to `Update-Arena-Dist` on line 5 rebalances the buckets according to the user's algorithm. We discuss our implementation of `Update-Arena-Dist` below. Finally, line 6 returns a new block to the user.

```
7  (define (Update-Arena-Dist old-arena)
8   (let ((new-arena
9    (optimum-ref optimal-arena
10    (minimize mem-usage
11     ...
12     (assert (>= optimal-arena
13                 (... history derate)))
14     (for ([b (range bucket-count)])
15      (if (ref adjust-bucket b)
16       (assert (... derate outliers))
17       (assert (... derate)))))))
18   (if (Converges? new-arena old-arena)
19    (Set-Arena new-arena)
20    (Update-Arena-Dist new-arena))))
```

Figure 3. Allocate example: `Update-Arena-Dist`

A straightforward way to rebalance buckets is to use heuristics such as high water mark or average value. Most heuristics may not be optimal overall, and some might not be easily specified.

Our implementation (Figure 3) adjusts the number of slots within each bucket via an optimization problem. The solution to this problem redistributes slots from an underused bucket to an overused one within the arena. It passes values back and forth cyclically until finally settling on a result that satisfies both host and solver. Note that we make a number of simplifying assumptions here—a real implementation must handle all the complexity of a real system.

In this example, we use three new features—`for`, `ref`, and functions over vectors. Iteration `(for ([i (range n)]) e)` can perform some computation `e` for index `i` from `0` through `n`. Vector indexing `(ref v k)` retrieves the `k`th index of the vector `v`. Both the index `k` and the upper bound `n` can be concrete values or symbolic variables. We discuss these in Section 4.

Line 7 defines `Update-Arena-Dist`, a recursive function that calculates a new arena by minimizing the total memory according to various constraints until it converges. On line 10 we cross the host-solver boundary via the `minimize` expression, going from normal functional semantics to Scimitar's solver semantics. Note how the syntax does not change across the transition, allowing programmers to focus on the semantics of their problem. See Section 3.1 for more details. This `minimize` expression introduces the symbolic `optimal-arena`, `mem-usage`, `derate`, and `adjust-bucket` variables. The result of the expression minimizes the total memory used by the new arena, while retrieving the arena itself. We omit the first few constraints for brevity, but intuitively they establish the relationship between the total memory and the new arena. Conceptually, these constraints flow the minimization of the memory usage down to the individual buckets, weighted by their sizes. The remaining constraints make up the core heuristic, which relies on the two symbolic variables `derate` and `adjust-bucket` to incorporate weighted allocation trends into the current arena via hysteresis.

The implementation of that heuristic uses the three features mentioned above. We determine `derate` dynamically by looping over the boolean vector `adjust-bucket` (up to the parameter `bucket-count`), starting on line 14. We access each bucket one at a time to determine the manner in which `derate` is constrained. The number of true buckets is upper bounded by a parameter set by the system. The conditional considers both derating possibilities simultaneously. If true for a particular bucket (line 16), its derating factor is constrained by a call to a function, including some `outliers` parameter known to the system. If false (line 17), `derate` is constrained by some different function that does not depend on the `outliers`.

The interaction between the two variables is subtle, and it may be the case that attempting to optimize the new arena according to the hysteresis criteria may fail some non-linear allocator convergence criteria. The algorithm transitions on line 18 back to the host semantics, then tests whether the new and old arenas converge. If the new arena has converged, then on line 19 the algorithm sets the arena and returns. Otherwise (line 20) it recurses with the new value. A real implementation would schedule this on some cadence instead of recursing, and after convergence to the steady state distribution would presumably switch to some cheaper but more approximate algorithm such as high water mark.

Note that unlike Figure 1, this recursion uses host semantics. Rather than creating a single optimization problem, host semantics generate a series of optimization problems executed one at a time. This is made easy by the design of Scimitar's virtual machine, which directly integrates continuations, easily allowing for such chaining (see Section 3.3). We discuss recursion using solver semantics in Section 4.2.

3 Functional Language

Figure 4 shows Scimitar's source language. The language includes variables x; values v; a letrec $f \Leftarrow e1$ in $e2$ expression that defines a function f, whose body $e1$ is always a lambda; lambdas $\lambda x \,.\, e$ and function application $e\ e$; conditionals if e then e else e; sequences of expressions $e\ ;\ \ldots\ ;\ e$, executing each expression for the constraints it might introduce and returning the result of the final expression; loops for $x \Leftarrow e$ do e that iterate through a tuple, assigning each tuple element in turn to x and evaluating the loop body; a native summation operation sum $x \Leftarrow e$ of e that is more efficient that summing via a loop; addition $e + e$; multiplication $e \cdot e$ over a real and a scalar or binary variable; vector indexing ref $e\ e$; and tuples $(\ e\ ,\ \ldots\ ,\ e\ )$.

$$
\begin{array}{rcl}
e & ::= & x \mid v \mid \mathsf{letrec}\ f \Leftarrow e\ \mathsf{in}\ e \mid \lambda\, x\, .\, e \mid e\ e \\
& \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid e\ ; \ldots ;\ e \\
& \mid & \mathsf{for}\ x \Leftarrow e\ \mathsf{do}\ e \mid \mathsf{sum}\ x \Leftarrow e\ \mathsf{of}\ e \\
& \mid & e + e \mid e \cdot e \mid \mathsf{ref}\ e\ e \mid (\ e\ , \ldots ,\ e\ ) \\
& \mid & \mathsf{assert}\ C \mid \mathsf{minimize}\ o\ e \mid \mathsf{optimum\text{-}ref}\ x\ e \\
C & ::= & e \preceq e \\
v & ::= & () \mid n \mid \alpha \mid \langle\ e\ , \ldots ,\ e\ \rangle \mid \mathcal{P} \\
o & ::= & x \mid n \mid \alpha \mid o \cdot o \mid o + o
\end{array}
$$

$$
\begin{array}{ll}
x, y \in \mathsf{vars} & n \in \mathbb{Z} \\
f, g \in \mathsf{funcs} & \alpha, \beta \in \mathbb{R} \\
\mathcal{P} \in \mathsf{prims} &
\end{array}
$$

Figure 4. The Scimitar source language

The language also includes the assertion form assert C, which generates the constraint C. Without loss of generality, in the grammar we write constraints as $e \preceq e$ where, as is standard in the optimization literature, $\preceq$ stands for $\geq, =,$ or $\leq$. To specify the optimization goal, the language has *solve expressions* of the form minimize $o\ e$, where the objective o indicates the expression to be minimized under the constraints generated in the body e. Objectives o are a syntactic subset of expressions. The result of a minimize expression is a *solution record* for all the variables mentioned in the objective. The form optimum-ref $x\ e$ returns the solution for x from the solution record e. Note that while the grammar here is limited to minimize, our implementation also supports a maximize solve expression.

Values in Scimitar are unit, numbers, vectors, and primitives. As discussed in Section 7, one deviation from the usual functional language semantics is that Scimitar does not support general lists, with list syntax instead denoting fixed-width tuples. *Primitives* $\mathcal{P}$ are native optimization problems stored in the solver's matrix format.

As stated previously, programs in this language are sent to the functional language compiler, which outputs code in O, the optimization problem language. O is (almost) a strict subset of the grammar presented in Figure 4. For a complete formalization, see Appendix B.

3.1 Semantics

Scimitar uses two sets of semantics: *functional host semantics* and *functional solver semantics*. Scimitar crosses this *host-solver boundary* via solve expressions.

Top-level code outside of solve expressions has the standard scheme-like semantics. In Figure 5, we give a specification of the functional host semantics' minimize and optimum-ref rules. The Solve rule minimizes some objective o subject to the constraints induced by the expression e to produce the solution record value v. The user supplies the objective o in a form that Scimitar can compile down to the equation $o = \sum_{i=1}^{n} \alpha_i \cdot y_i + \alpha \cdot y + \beta$.

To evaluate e, we switch to the functional solver semantics. Instead of executing e as in the host semantics, the solver

$$\frac{\begin{array}{c} o = \sum_{i=1}^{n} \alpha_i \cdot y_i + \alpha \cdot y + \beta \\ \min \alpha_1 \cdot y_1 + \ldots + \alpha_n \cdot y_n + \beta; C \vdash e \rightsquigarrow y \\ [\![o]\!] = c^T x \qquad [\![C]\!] = Ax \preceq b \\ \text{solve}(\operatorname{argmin} c^T x \text{ s.t. } Ax \preceq b) \to x^\star \qquad [\![v]\!] = x^\star \end{array}}{\text{minimize } o\ e \Downarrow v} \textsc{Solve}$$

$$\frac{e \Downarrow v}{\text{optimum-ref } x\ e \Downarrow v[x]} \textsc{OptRef}$$

Figure 5. The solve rules of the functional host semantics The judgment form here is the usual big-step $e \Downarrow v$

semantics determine an equivalent optimization problem, if it exists, via angelic nondeterminism [Bodík et al. 2010]. The solution to this problem is an assignment for the objective that, when substituted into the body of e, yields a program that correctly obeys the host semantics. The judgment form used in the solver semantics is trace-based. The relation $e \rightsquigarrow y$ states that the solver can only reason about the operation e in a context where the solver is aware of some result variable y. The solvability of this relation is contingent on the existence of some equifeasible MILP problem that minimizes an objective subject to constraints C over the objective and result variables. For such an equifeasible MILP problem to exist, the variables and constants in its objective and result must be a superset of those requested by the user-supplied objective o. For a complete discussion of the functional solver semantics, please see Appendix A.2.

The Scimitar objective and constraints are converted to the solver's vector and matrix format $\operatorname{argmin} c^T x$ s. t. $Ax \preceq b$ known as *standard form*. Standard form minimizes the function $c^T x$, where x is a vector comprised of all program variables $\langle y_1, \ldots, y_n, y\rangle$ for the solver to decide, and c is the given coefficient vector $\langle \alpha_1, \ldots, \alpha_n, \alpha\rangle$ (we ignore β since it doesn't impact the argument values). In the constraints $Ax \preceq b$, the term A is the matrix of coefficients, and b is the vector of bounds. Once assembled, this standard form problem is submitted directly to the solver.

The result is the vector of the optimal values of x, known as the *optimal point* $x^\star$. We convert $x^\star$ into the opaque Scimitar solution record value v, whose members are accessed by OptRef.

3.2 Types

Scimitar includes a type system built around vector shapes and value sets. The language includes a variety of types, but they are ultimately converted into $\mathcal{I}^{\mu}$ (vectors of shape μ over some interval $\mathcal{I}$). Of particular interest are the set of reals $\mathbb{R}$ and the set $\{0, 1\}$. We discuss types in detail in Appendix A.1.

3.3 Virtual Machine

Scimitar's virtual machine uses a CPS execution model internally. The compiler breaks code blocks into parameterized continuations, which it stores in a table with an associated key. The virtual machine loop looks up a key, loads the continuation, applies parameters, and finally executes it. When complete, each continuation returns the key for the next one to execute.

Because Scimitar supports first class functions, optimization problems can take advantage of this execution model by using continuations to direct control flow dynamically across the host–solver boundary (for more on this topic, see Sections 2.2 and 3.1).

3.4 Solver Awareness

To use Scimitar, users must sometimes be aware of the solver's behavior and configuration. Solvers themselves have limits, like allowable solver values and finite problem size.

For example, different solvers have different numeric limits internally, which restricts the algorithm's precision and the values it can use. To control this behavior, Scimitar users may have to configure the largest and smallest allowed values before compilation to ensure the solver can handle all values needed by the program. For example, the *contradict* benchmark (Section 5) with the default bounds is infeasible using Gurobi. By setting the bounds within 6 orders of magnitude from largest to smallest value, it solves correctly. In general, multiplication and conditional constraints may introduce such precision issues. To avoid them, programmers must take care that values used in those expressions have tighter bounds on their type (see Section 3.2).

Solver problems must be finite, which comes into play with constant propagation in loop unrolling. If a loop's bounds can be reduced to constant numeric values via constant propagation, Scimitar unrolls the loop precisely that many times, propagating the loop variable's value for each iteration. However, if the loop's bounds cannot be statically determined, e.g., if they use unknown optimization variables, then Scimitar can not know how much to unroll. It must still finitize the program though, so it unrolls the loop up to a user supplied loop unroll bound parameter. The user has to be sure that Scimitar is configured with an unroll limit that is sufficient to solve the problem. They must also take care that the limit is tight, because if it is too high, this can dramatically increase the solve time. The user must be aware of these limitations when designing their programs.

4 Encoding Scimitar to Constraints

Scimitar uses a range of techniques and strategies to encode high-level program features. In this section, we demonstrate some of the most important ones. We also discuss some of the areas where Scimitar must take special care to avoid potential pitfalls.

$$
\begin{array}{lcl}
[\![x \cdot y]\!] & \geq & x^l \cdot y + x \cdot y^l - x^l \cdot y^l \\
[\![x \cdot y]\!] & \geq & x^u \cdot y + x \cdot y^u - x^u \cdot y^u \\
[\![x \cdot y]\!] & \leq & x^l \cdot y + x \cdot y^u - x^l \cdot y^u \\
[\![x \cdot y]\!] & \leq & x^u \cdot y + x \cdot y^l - x^u \cdot y^l
\end{array}
$$

$$
\begin{array}{lcl}
[\![\text{if } c \text{ then } b_t \text{ else } b_f]\!] & = & c \cdot b_t + (1 - c) \cdot b_f \\
[\![\text{if } c \text{ then } b_t \text{ else } b_f]\!] & = & \forall\, r_{t_i} \in \text{free}(b_t). \\
 & & \quad v_{t_i} = \text{if } c \text{ then } r_{t_i} \text{ else } d_{t_i} \\
 & & \forall\, r_{f_j} \in \text{free}(b_f). \\
 & & \quad v_{f_j} = \text{if } c \text{ then } d_{f_j} \text{ else } r_{f_j} \\
 & & b_t[r_{t_0} \mapsto v_{t_0}] \ \ldots \\
 & & b_f[r_{f_0} \mapsto v_{f_0}] \ \ldots
\end{array}
$$

$$
\begin{array}{ll}
[\![\text{ref } (v_0 \ldots v_n)\ y]\!] = \sum_{i=0}^{n} b_i \cdot v_i \quad \text{s.t.} & y = \sum_{i=0}^{n} i \cdot b_i \\
\qquad\qquad 1 = \sum_{i=0}^{n} b_i & \forall\, 0 \leq i < n \,.\, b_i \in \{0, 1\}
\end{array}
$$

Figure 6. Encodings of various useful language features

4.1 Exact Encodings

Figure 6 gives the encodings used in Scimitar for several key language constructs (described in Section 3): variable multiplication, conditionals, and dynamic indexing. Figure 7 presents the definitions of two libraries: Boolean algebra and variable comparison.

Variable Multiplication. Unfortunately, linear programs cannot multiply two variables, nor can MILP programs without special constraint formulas that create a relaxation of the expression. For this discussion, we break apart variable–variable multiplication into three cases: general continuous–continuous, binary, and integer.

The first is not possible in MILP, and requires a more powerful solver (see Section 7).

Scimitar does implement multiplication by a binary variable, which uses the standard encoding using McCormick envelopes [Dombrowski 2018]. For the exact encoding, see the top set of equations in Figure 6. This encoding creates a convex relaxation that simulates multiplying a binary variable with any other integer or continuous variable, which approximates the original nonlinear function. This method relies on knowing the upper and lower bounds (denoted by superscript u and l, respectively) of each variable. Scimitar carries these bounds in the variable's type, which makes implementing this translation simple (see Section 3.2). Experts may find introducing McCormick envelopes simple for small programs, but as the program grows it becomes increasingly difficult to track and maintain them.

While possible, multiplication by integer variables is more involved, and Scimitar does not currently support this.

If-Then-Else. Conditionals are implemented in two cases, shown in the middle of Figure 6.

In the *simple case*, Scimitar encodes the guard as an indicator variable c that is multiplied by the branches b_t and b_f, thus effectively "disconnecting" the other branch. By plugging true and false into c, we can see that the corresponding branch value is returned. The simple case is used for conditional formulas that does not include any constraints.

Otherwise, in the *constraint case* Scimitar uses a more complex encoding. As with the simple case, to translate the constraints, we must disconnect each branch depending on the result of the guard. This is more complicated here because asserts must only activate when their corresponding branch is selected. We disconnect branches by replacing their variables with dummies. First, we collect the *real* free variables r_x from each branch, where x ranges over the free variables r_{t_i} and r_{f_i}. For each variable, we create a *dummy* variable d_x. We then select between these one by one using the encoding from the simple case, assigning each to the *used* variable v_x. Finally, the branches are encoded but substituting the real variables for the used ones.

This approach allows the solver to satisfy each branch's constraints with variable assignments that do not impact the rest of the problem. The dummies are unconstrained slack variables that "float" without impacting the computation. Thus the active branch uses the real variables, while constraints in the inactive branch have no effect[1].

Dynamic Indexing. Dynamic vector and matrix indexing ref $(v_0 \ldots v_n)$ y is the process of selecting a specific vector element v_k using a variable y as the index. Dynamic indexing is required, for example, in a recursive function with an argument representing the index into a vector, as discussed in Section 4.2. Because the index is a variable, the compiler does not know in advance how to access the vector, and must leave it to solve time. This is not a simple operation, in contrast to selecting an element using a constant, which is trivial. The implementation of dynamic indexing is shown at the bottom of Figure 6.

First, the encoding needs one-hot indicator variables b_i for each index v_i in the vector. Only the indicator b_k is nonzero, so the sum of all $b_i \cdot v_i$ is equal to the corresponding vector entry v_k. The variable y is translated into the one-hot encoding by scaling each indicator by its corresponding index. For example, to select the third index, the third indicator variable b_3 is multiplied by three. This will then select the third vector element v_3 as the result. The encoding is constructed this way so that both constraints on v_i and y affect the value of the ref, and so that constraints on the ref affect the values of the v_i and y.

Boolean Algebra. The encoding of Boolean algebra is straightforward, as shown in the top set of equations in Figure 7. The Scimitar library implementation uses one for true and zero for false, and each Boolean operation can

[1]With one exception: if the programmer includes unsatisfiable constraints such as $0 = 1$ in a branch that is not taken, the problem will be infeasible overall.

$$\begin{aligned}
\text{true} &= 1 \\
\text{false} &= 0 \\
\text{not}(b) &= 1 - b \\
\text{and}(b_1\ b_2) &= b_1 \cdot b_2 \\
\text{or}(b_1\ b_2) &= b_1 + b_2 - b_1 \cdot b_2 \\
\text{xor}(b_1\ b_2) &= b_1 + b_2 - 2 \cdot b_1 \cdot b_2
\end{aligned}$$

$$\begin{aligned}
&\text{cmp}(n\ m\ c_<\ c_=\ c_>) = \\
&\quad \{\ 1 = c_< + c_= + c_>, 0 = c_< \cdot (m - n - \epsilon), \\
&\quad\ \ 0 = c_= \cdot (n - m),\ 0 = c_> \cdot (n - m - \epsilon)\}
\end{aligned}$$

Figure 7. Encodings of useful library functions

be modeled using simple sums and multiplications of their arguments. By plugging true and false into each formula, we can see that the corresponding truth table is satisfied.

Variable Comparison. Testing variables for equality or inequality is a necessary operation for any numeric computation system. SCIMITAR's library encodes it using one-hot indicators $c_<$, $c_=$, and $c_>$, each of which is true when one input is less than, equal to, or greater than the other, respectively. The encoding of this test relies on some parameter ϵ, as shown in the cmp equations at the bottom of Figure 7. Because they are mutually exclusive, if $c_=$ is true, it must be the case that $n - m$ is zero. Otherwise, this difference must be at least ϵ away in one direction or the other (as the smallest value in our system, we do not worry about differences smaller than ϵ).

4.2 Approximate Encodings

Figure 6 omits some important encoding techniques for language constructs. One thing to note is that unlike the exact encodings, in certain cases these techniques can lead to infeasible problems, and some do not achieve the optimal result. The user must pay special attention to their use in practice, as programmers often have sideband information that is not encoded directly into the program and that is impossible to determine programmatically. Avoiding these bad cases may require tweaking some runtime parameters, such as inlining and unrolling bounds, as we discuss next.

Inlining. An optimization-aided language must inline all function calls, both normal and recursive. As with traditional languages, function inlining directly replaces a function call with the function body. Unlike traditional languages, optimization problems are unable to reuse code. Each call to a function must be translated into a separate copy of that function's constraints to allow the solver to freely set its variables independently of the other calls to the same function.

Optimization problems are intrinsically bounded. Finitizing a recursive function is challenging because it can only be inlined up to some depth limit parameter supplied by the user at compile time, and is unable to be invoked beyond this depth. When the limit is reached, SCIMITAR compiles in a sentinel with an associated path condition. Given a sufficient depth limit, this path condition will not be met at solve time because the solver will be able to reach the program's base case. The sentinel is triggered when the solver has no choice but to meet its path condition, which conditionally makes the program infeasible (as discussed above in Section 3.1). As a consequence of this, recursion without a base case will always yield infeasible results.

Loop Unrolling. SCIMITAR finitizes loops by unrolling them. We discuss two cases of loops: known bounds and unknown bounds. SCIMITAR can completely unroll finite loops because the number of iterations is known at compile time. If it is impossible to calculate the range of iteration, such as when the upper bound of the type uses a solver variable, compilation falls back on a translation to a recursive function using an index parameter. This produces a sequence of index-guarded loop bodies up to a user-supplied unroll limit parameter. As with inlining, the drawback of this approach is that the problem will be infeasible when the solver is unable to find a solution within the number of unrolled iterations.

4.3 Other Considerations

Finally, to avoid explosion of the size of the compiled program, it is critical to optimize the whole-program representation. This includes using compact code and data representations, type information, compiler passes such as constant propagation, and constraint normalization.

Data and Problem Representation. It is critical to have an efficient representation for optimization problems and their data. High level languages are highly compositional, with most terms containing several subterms. During the translation to solver-level constraints, this can quickly lead to large optimization problems, as translated subproblems will constantly have to be linked up using conjoining constraints, introducing a constant number of extra constraints per subterm. Because this is a frequent operation, the speed of concatenating constraints is critical. Thousands of intermediate constraints may be constructed, so this must take as little time as possible.

Unavoidably, the compiler may have to inspect or modify the contents of a given solver constraint or value, e.g., when selecting an index, merging two constraints, splitting one constraint into multiple, or scaling a value. An efficient representation can have an asymptotic speedup for these operations. In SCIMITAR, we elected to represent optimization problems as normalized sparse matrices augmented with type metadata. Due to the compositionality discussed above, most constraint rows reference very few variables, and those variables are often only used in adjacent rows. Because the output problem generated by our compiler is very sparse and clustered around the matrix's diagonal, this is a very efficient representation.

Types. Scimitar's types are invaluable for several reasons (see Section 3.2). Firstly, we must track the type of data for the sake of the solver, whose internal representation requires variables to be bounded. As usual, it is important to verify whether operations are even permitted on certain variables. Without types, it is impossible to implement the encodings we have mentioned, e.g., McCormick envelopes and dynamic vector indexing, as they need to know the bounds and shapes of their arguments. The approximate encodings we mentioned above would also be impossible without tracking variable types, as we need them to check the path condition of the sentinel. Finally, data flow and control flow across the host-solver boundary would not be possible without types, since the compiler is not designed to introspect and infer the types of runtime data. Even if the functional language values were themselves untyped, we must know the relationship between the functional language representation and the optimization language representation in order for values to cross the boundary. We discuss types in detail in Appendix A.1.

Compiler Passes. The Scimitar compiler is comprised of several successive passes. Transformations such as constant propagation are critical to minimize the number of constraints. Because assembling the final standard form problem is the most expensive step of compilation, the fewer constraints that reach that phase, the faster the program will compile. By eliminating or combining redundant and coupled constraints, we can offset changes later in the pipeline where additional constraints must be generated, such as in the case of McCormick envelope expansion. There is a trade off, however, because the time spent performing preprocessing may actually exceed the time spent in the solver. We found that in practice, most passes executed in a matter of microseconds, or at worst, a few milliseconds. This is in contrast to assembling the standard form problem, which was the compiler performance driver, and represents the biggest opportunity for future performance improvements; see Section 5.

Constraint Normalization. One particularly important requirement for the compiler is to reformulate the constraints into a normalized representation by successively rewriting terms into lower level representations and guaranteeing that some terms only appear in certain positions, like requiring the left subterm of multiplication to be a number. It is important to eliminate terms as early as possible, thereby requiring fewer redundant decisions. This is one reason we separate the Scimitar grammar from the optimization language grammar. There are even cases where one constraint must be split into two to create this streamlined representation. In addition to performance, a side benefit of normalization is that it helps in simplifying subsequent compiler passes.

4.4 Pitfalls

While the encodings discussed above present opportunities to support a diverse selection of high level language constructs, there are many challenging details that had to be carefully overcome or avoided to implement Scimitar. We briefly discuss two of these issues.

Types. As powerful and necessary as types are in Scimitar, they are not trivial. The compiler infers the types of variables and expressions, which is nuanced. The core feature of the type system is the shape and bounds of vectors as mentioned in Section 3.2, and type information is required to correctly lower many operations constraints.

Vectors of different bounds and shapes must often interact. This could be a problem in a system that required strict type equality, as many expressions would not type. Because vector types in Scimitar form a partially ordered set over their bounds and shapes we can accommodate such interaction by performing subtype inference.

In some corner cases, such as general variable multiplication or indexing a vector by the index of another vector, Scimitar cannot infer the type correctly, because the type system lacks the richness to cover these cases. Scimitar currently requires type annotation for those cases, and will produce a type error if they are missing.

Higher Order Functions. Compiling higher order functions in Scimitar requires careful handling of function arguments. There can be conflicts between supplying a function argument and actually applying that parameter in the body of the higher order function. These conflicts arise with functions that can not be fully inlined. In cases such as returning a function from another function, Scimitar is not able to decide how that resulting function can be invoked. As a result, Scimitar's implementation of first class functions is incomplete. We leave this to future work.

5 Evaluation

To demonstrate Scimitar's features and explore its capabilities, we developed several benchmarks:

- *logistics* is a traditional logistics example of optimizing for profit.
- *pipes* is a simple network flow problem demonstrating basic language features. This and *logistics* show Scimitar's applicability to classic optimization problems.
- *malloc* is a full implementation of the program sketch described in Figures 2 and 3.
- *recitation* minimizes the number of sections required to adequately serve the students from a class.
- *contradict* tests a conditional that contains a contradiction.
- *bounce* recurses on a simple heuristic that bounces back and forth until converging.
- *sum-to-n* is the problem presented in Figure 1.

- *imp* is a complete implementation of Winskell's Imp language, which demonstrates Scimitar's full modeling power.

We discuss the design of each benchmark below and give its size in terms of source lines of code (*sloc*), including the extra Racket support code required to decide program properties at compile time such as input data formatting. For each benchmark, we also report Scimitar's performance in terms of compile and solve times. For several benchmarks, we wrote a corresponding problem directly in the optimization language $\mathcal{O}$. In these cases, we compare the performance of the Scimitar-generated problem encoding to the directly written version. Most programs compile faster in $\mathcal{O}$, but the difference in performance is unpredictable. For several problems, we also compare performance against Rosette versions of the benchmarks. We use Rosette's `optimize` query, which depends on Z3's MaxRes algorithm. The performance differences are split between time spent encoding programs and time needed by the underlying solver. Overall, Scimitar's compile times are two orders of magnitude slower than Rosette, but its solve times are an order of magnitude faster.

Our benchmark code is available along with the Scimitar implementation [Bragg et al. 2024].

5.1 Benchmarks

Logistics. One of the classic domains addressed by optimization solvers is logistics problems. We implemented an example logistics problem that optimizes and returns the maximum profit expected from some commercial warehouse and trucking enterprise.

Originally, we implemented the benchmark in the optimization language—at the time, Scimitar source language was not fully developed. We then continued to co-develop the source language and the example. This back-and-forth drove the design of Scimitar, and caused us to add support for dynamic loop unrolling, dynamic recursion inlining, and dynamic vector indexing. Scimitar also influenced the implementation of the benchmark. Because solve-time conditionals are central to Scimitar, as we ported to the functional language we replaced operations that were implemented using MILP encoding techniques with structured code using `for` and `if`. This led to simpler and more intelligible code. To see the exact differences between the two versions, we refer the reader to the benchmark implementations in the Scimitar repository.

Overall, we found that Scimitar source language code is much easier to update and modify. Although the raw number of constraints generated by our compiler for a given problem is a constant multiple greater than the number in the manual version, the Scimitar version is more maintainable because as the input program grows linearly in size, the number of constraints in the compiler output (and the manual version) grows quadratically. Because of the design of this benchmark, most constraints in both the Scimitar and $\mathcal{O}$ versions are highly redundant, and some are similar but have nuanced differences. This can make it difficult to maintain the $\mathcal{O}$ version if the data changes. Furthermore, the Scimitar version can be much more easily changed, e.g., to also optimize the number of trucks, while in a hand-written version this would require drastic changes.

```
1  (optimum-ref sink-out
2   (maximize sink-out
3    ; omitted source and sink constraints
4    (for ([j (range (length num-is))])
5     (assert ; junction inflows = outflows
6      (= (sum ([i (range (ref num-is j))])
7          (ref pipes (ref j-is `(,i ,j))))
8         (sum ([i (range (ref num-os j))])
9          (ref pipes (ref j-os `(,i ,j)))))))
10   (for ([i (range (length pipes))])
11    (begin ; pipe flows in allowed range
12     (assert (<= (- (ref pipes i))
13                 (ref pipe-flows i)))
14     (assert (<= (ref pipes i)
15                 (ref pipe-flows i)))))))
```

Figure 8. The *pipes* problem

As a demonstration of the ease with which we can change the higher-level benchmark, we measured two versions of the problem, one smaller and one larger. The *logistics-s* program has 1 product, 2 cities, 1 road, and 1 truck, while *logistics-h* has 4 products, 4 cities, 4 roads, and 8 trucks. As stated previously, this increase in program size results in a disproportionately larger number of rows and variables in the output, and a greatly increased solve time, as shown below in Table 1.

Our implementation of *logistics* is 145 Scimitar sloc, with supporting Racket code of 77 sloc.

Pipes. Figure 8 shows an excerpt from the Scimitar code for *pipes*, which calculates the maximum flow through a pipe network, a classic optimization problem. For the flow to be valid, the inflow and outflow of every junction must balance (lines 4-9) given each pipe's capacity (lines 10-15). Additional constraints matching source and sink flows have been omitted. Our implementation resembles the usual formal problem statement, but is written such that the programmer need not directly encode each constraint and couple it to the data. A modest amount of extra support code is needed to preprocess the data from a graphlike representation into the format that the algorithm expects. After the Scimitar program is compiled to the optimization language, the constraints generated are similar to a hand-written version of the problem.

This basic problem demonstrates Scimitar's capabilities on a convex problem with no integer variables. The Scimitar code for this traditional optimization problem is simple to implement, easy to understand, and straightforward to verify.

```
(:= y 0)
(:= x 0)
(while (<= y 5)
  (:= x (+ x y))
  (:= y (+ y 1)))
```

Figure 9. Example of an Imp program

Pipes also serves as a good benchmark to compare with Rosette since the two implementations are virtually identical. The only significant difference is the declaration of symbolic variables, which is explicit in Rosette and implicit in Scimitar.

The *pipes* benchmark is 21 Scimitar sloc, with 113 supporting Racket sloc.

Malloc. The *malloc* benchmark shows a more real-world utility, something a developer might want to write as a part of their program. This benchmark is a version of the memory allocation example presented in Section 2.2. The original example serves as framework that an author of a memory management system could implement using their own criteria. The snippet shown in Figure 3 is incomplete, and serves to explain the features of Scimitar. The benchmark implementation we measured uses a different heuristic that is more complex and fleshed out than the one in Figure 3. *Malloc* also demonstrates interaction across the host–solver boundary, where variable optimums in one iteration become parameters in the next iteration. The *malloc* benchmark is 29 Scimitar sloc, with 71 supporting Racket sloc.

Contradict, Bounce, Sum-to-n. These benchmarks are toy examples that we used to demonstrate the basics of Scimitar. The *contradict* benchmark attempts to minimize the expression `(if x (assert (= x 0)) (assert (<= x 1)))`. The true branch introduces a contradiction, and Scimitar correctly determines x to be false. In *bounce*, we recurse on the output of a toy heuristic, passing the previous result into the next round until the values converge. We give the code for *sum-to-n* in Figure 1.

Recitation. This benchmark minimizes `rec-count`, the number of sections required to adequately serve the students in a class (thereby reducing the number of TAs, rooms, etc.). The problem guarantees that all registered students are assigned a section, and that recitations are only scheduled if they meet a minimum registration count. Recitations are modeled using symbolic variables for the possible section slots that could be scheduled and their attendance. To solve this, the program uses `rec-count` as the upper loop bound (as discussed in more detail in Section 4.2). This construction simplifies the representation, because the relationships between the students and sections can be enumerated without worrying about the number of sections needed or precisely which ones are selected. The *recitation* benchmark is 43 sloc for both Scimitar and supporting code, respectively.

Imp. Winskell's Imp language is a minimalistic imperative language supporting conditionals, loops, and basic operations on numbers and booleans. For example, Figure 9 shows a simple iterative summation program in Imp, akin to Figure 1. The code initializes two variables `x` and `y`, then loops incrementing `y` and adds that to `x` until `y` is greater than 5.

We developed a compiler that translates an Imp source program to Scimitar source. We chose this example to illustrate the versatility of Scimitar to handle more complex iterated domains. Note that this is unlike other benchmarks, which are written directly as Scimitar source programs. The compiler works by splitting Imp source into basic blocks. Each basic block is compiled to a continuation that contains a `minimize` expression. As the compiled program executes, control flow from one basic block to another corresponds to one continuation calling another in the virtual machine.

Mutable state is represented as an environment that gets passed from one continuation to the next. For example, for Figure 9, the environment is a pair containing `x` and `y`.

Note that the compiler deliberately does not unroll loops in Imp. In theory, loops like the one shown in Figure 9 could be unrolled. However, we wanted the compiler to support full Imp semantics, including non-termination of loops such as `(while true skip)`. Generally, functionality which can not be shown to terminate or is unbounded in size can not be finitized by Scimitar into an optimization problem; doing so requires stepping back into the host language.

The Imp compiler is 475 sloc. Unlike other benchmarks, the implementation of the compiler intermixes Scimitar and Racket code together, with no simple breakdown between the two.

5.2 Solver

For our evaluation, we used the popular off-the-shelf Gurobi MILP solver [Gurobi Optimization, LLC 2023]. We chose Gurobi because of its versatile and rich API and excellent performance. The sparsely encoded matrix and vector representation our compiler uses is easily translated to Gurobi's expected format, and from there it is loaded using the solver's FFI. Scimitar performs this translation immediately before the solving step in the virtual machine. We attribute the time spent in this translation and loading step to compile time.

5.3 Results

Table 1 gives the median run time performance in milliseconds of each benchmark program and the variable and row count for the compiled code. All measurements were taken on a 3.2 GHz AMD Ryzen 5 1600 system with 32 GB of RAM using Racket's `current-inexact-milliseconds` function.

Note that the comparisons of Scimitar to Rosette are, to a certain extent, comparing the efficiency of Gurobi and Z3. Z3 implements primal simplex and MaxRes [Bjorner 2022], while Gurobi uses branch-and-bound [Gurobi Optimization, LLC 2023] (with relaxation to simplex), which we expect to be

Table 1. Measurements in seconds for Scimitar, optimization language, and Rosette programs

			Scimitar		O		Rosette	
	vars	**rows**	**compile**	**solve**	**compile**	**solve**	**compile**	**solve**
pipes	161	179	34.9 ms	0.60 ms	0.60 ms	18.8 ms	0.45 ms	0.0 ms
logistics-s	268	293	53.5 ms	2.0 ms	1.4 ms	21.3 ms	2.2 ms	30.8 ms
logistics-h	10884	14026	478.7 ms	1032.9 ms	674.3 ms	112.6 ms	66.6 ms	>20 m[2]
contradict	158	229	33.3 ms	1.0 ms	0.52 ms	2.3 ms	0.08 ms	25.2 ms
sum-to-n	11817	17990	1417.2 ms	54.7 ms	—[1]	—[1]	>20 m[2]	—
bounce	1858	2538	130.5 ms	12.9 ms	—[1]	—[1]	1.1 ms	51.1 ms
recitation	34846	52514	2190.1 ms	615.2 ms	—[1]	—[1]	49.7 ms	125490.0 ms
malloc	227	239	128.6 ms	4.2 ms	—[1]	—[1]	8.7 ms	189.4 ms
imp-s	8171	10158	201.4 ms	16.5 ms	—[1]	—[1]	—[1]	—[1]
imp-h	11068	13074	689.6 ms	16.6 ms	—[1]	—[1]	—[1]	—[1]

[1] These benchmarks do not have a corresponding non-Scimitar version.
[2] This benchmark exceeds a 20 minute time out.

faster. However, the benefit of using a dedicated optimization solver is also part of the benefit of using Scimitar.

The comparison of Scimitar to Rosette also measures the efficiency of our respective encodings. In both cases, this time is split between compile and solve time, though in different ways. Scimitar's encoding and high-level optimization are mostly performed at compile time, with lower-level optimization left to Gurobi. In contrast, in Rosette, the encoding happens at compile time, and the optimization happens at solve time.

Table 1 compares our benchmarks written in Scimitar, in the optimization language O, and in Rosette. Generally, the performance bottleneck in Scimitar is the compiler, which compares unfavorably with the others. Conversely, the solve time of Scimitar is dramatically better than in Rosette. Surprisingly, the solve time of the optimization language version exceeds Scimitar in some cases, which implies that the compiled version is more efficient than one that would be written directly. This is because solver performance is unpredictable, and small changes sometimes greatly affect running time.

Examining the results in more detail, we make several additional observations. The performance of *logistics-h* as compared to *logistics-s* degrades non-linearly. The increase in size by a factor of over 40× is because there is simply much more data to account for. The benchmark's complexity scales with the scaled sum of products of these parameters. As a result, the problem has 14026 constraints and 10884 variables, many of which are binary. Unfortunately, while it is often possible to relax binary variables to continuous ones, in this case it is not possible because all of them are indicator variables. In this benchmark, performance is wildly different across the three versions. Both Scimitar and the optimization language version return correct results, but the latter version is over nine times faster. The Rosette version is virtually the same as the Scimitar version, but it exceeds a 20 minute time out, so we are unable to evaluate it. We do not know if this is an issue with Rosette or with Z3.

While Scimitar executes *sum-to-n* quickly, Rosette hits a 20 minute timeout while compiling. As such, there is no solve time measurement for this benchmark in Rosette.

The performance of *recitation* is over two orders of magnitude faster in Scimitar than in Rosette. We believe this is because the benchmark optimizes over a loop bound, which we encode efficiently.

While executing, *malloc* makes multiple solver calls, and the times presented reflect the sum total across all runs. Because of the nature of this design, we think this particular benchmark's performance would benefit disproportionately from compiler performance improvements.

The two *imp* benchmarks also make multiple solver calls, but unlike *malloc*, which only has a single problem that is solved repeatedly, *imp* changes between different problems dynamically. The sizes given are the sum across all generated problems. *Imp's* compiler performance reflects the high number of individual problems that a given *imp* program uses. Both *imp-s* and *imp-h* are similar. In fact, *imp-s* is basically a subset of *imp-h*. The size difference is linked to overall program size. Both benchmarks are large overall compared to our other benchmarks due to the difficulty of encoding an imperative programming language as a constraint problem.

It was not possible to replicate all Scimitar benchmarks using the optimization language. The use of some key Scimitar features would require reproducing large chunks of the output of the compiler. Additionally, Scimitar programs may solve problems successively, which the optimization language is incapable of. For the *imp* compiler, while it would be possible to write a similar optimization language program for a given Imp program that does not require these features, it would represent an enormous engineering effort, on the order of writing the original compiler. Likewise, we did not attempt to replicate *imp* using Rosette as this would require a significant engineering effort.

The performance of these benchmarks is clearly correlated with the number of variables and constraints[2]. One nice feature of our encoding is that, while these numbers seem to suffer from a blowup with increased number of variables v and rows r, growing at the rate of $O(v \cdot r)$, the encodings that SCIMITAR produces are normally very sparse, leading to an increase that is closer to $O(v + r)$.

In summary, while SCIMITAR's compiler is not the fastest overall, the improved solver speed gives evidence that a dedicated MILP solver compares favorably with the more general purpose SMT solver on optimization applications.

5.4 Analysis

Beyond raw solver performance, there are other differences that may influence the choice of one language over another.

While Rosette can make use of Z3's optimization capabilities, it remains a satisfaction-oriented language. For problems where that is sufficient, Rosette would be preferable, and will almost certainly outperform SCIMITAR. However, as we see here optimization is not its focus, and for such problems SCIMITAR will be more performant.

Another reason to use Rosette is its support for queries beyond what is discussed in this paper, including verification and program synthesis. Furthermore, the implementation of Rosette incorporates a rich subset of Racket's features, while the current implementation of SCIMITAR is less complete.

SCIMITAR and Rosette naturally model data differently in accordance with the underlying solvers, with SCIMITAR being built on top of primitive MILP concepts, while Rosette's data is restricted to types supported by SMT. Accordingly, data in SCIMITAR is built on top of procedures and vectors over scalar sets, while Rosette's core data types are booleans, integers, reals, bitvectors and uninterpreted functions. While both languages track the types of variables, SCIMITAR infers the types of variables and expressions, while Rosette requires type ascription and does not support typing of expressions.

Finally, the applications are quite distinct, and despite considerable overlap, they target different classes of problems. Those given by Rosette's developers include formal verification of systems such as JIT compilers, synthesis of GPU kernels, program repair, and model checking. On the other hand, SCIMITAR targets applications such as logistics problems, resource allocation, and design optimization.

6 Related Work

MILP Encoding. Popular encoding techniques are diverse [Braun 2012; Dombrowski 2018; GAMS Development Corporation 2022; Huchette and Vielma 2019, 2022; Pardalos 1988; Ruiz et al. 2012; Wikipedia contributors 2024] and implemented on a case-by-case basis in various MILP frontends. While SCIMITAR does not implement every technique listed, it aims to provide the programmer with an environment where they do not have to think about the details of encodings.

Embedded Optimization Problems. Systems such as cvxpy [Diamond and Boyd 2016], JuMP [Dunning et al. 2017] and Matlab [The MathWorks, Inc. 2020] embed mathematical programming within a general purpose language. These systems are focused on dynamically constructing an optimization problem piece by piece, not on modeling a general purpose language. In contrast, SCIMITAR is not an embedded language. Instead, it reasons directly about programs themselves, rather than constructing a problem out of pieces.

The classic constraint language is AMPL [Fourer et al. 1987]. Although featureful, it does not attempt to model high-level language features.

Unique Computing Paradigms. Rosette [Torlak and Bodík 2013] explored embedding SMT solvers in a functional programming language, and was an inspiration for SCIMITAR. Both are frameworks that use symbolic expressions translated to constraints then submitted to a solver via an angelic execution query. Both enable developers to write and reason symbolically about programs using high-level abstractions in the traditional functional style, including constraints over symbolic expressions. The modeling language matches the host language, so there is less cognitive load when transitioning across the host-solver boundary. While inspired by similar principles, Rosette is focused on verification and synthesis, while SCIMITAR is meant for optimization.

Differentiable programming [Abadi and Plotkin 2020; Bošnjak et al. 2017; Manzyuk 2012] is an approach to numeric programming that uses automatic differentiation [Baydin et al. 2018; Hascoët and Pascual 2013; Pearlmutter and Siskind 2008] to generate the derivative of a program for use in algorithms such as gradient descent. This is a parallel to SCIMITAR's approach—it directly applies an alternative interpretation of a program, expanding that program's semantics to include the result. To do this, it must syntactically analyze the input program to generate the derivative. Similar to SCIMITAR, this requires unique handling of conditionals, which are normally discontinuous.

We designed SCIMITAR as a standalone language because of the unique demands of the runtime system. The HANSEI [Kiselyov and Shan 2009] language is an example of using a similar approach to embed its special computing paradigm, probabilistic programming, while making use of host language features. As opposed to cvxpy and JuMP, the embedding is much more shallow, and HANSEI programs hardly look different than programs in a standalone probabilistic language. SCIMITAR cannot be similarly embedded because it must reason about global properties like control

[2]The exception being *malloc*, whose performance is skewed because it is compiled multiple times within a single execution.

flow and variable use that require an ability to reason about the implementation of every function.

Embedded Functional Program. Some recent efforts have been made to encode high-level language features using constraints. These languages compile programs in an imperative syntax to constraints, which allows for the possibility of deciding satisfaction or performing optimization, without requiring programmers to work directly in the constraint language.

BFDL [Braun 2012] is a high level language derived from Fairplay [Malkhi et al. 2004]. It offers encodings of several language features into a constraint system including non-recursive function calls, fixed iteration loops, conditionals, boolean algebra, integer arithmetic, comparison, user-defined structs, and constant index array access. This is vaguely similar to Scimitar's optimization language, except that BFDL is focused on constraint satisfaction, whereas Scimitar is meant for optimization.

CirC [Ozdemir et al. 2020] is a compiler framework for proof systems that uses ILP as one of its backends, and can use that to discover output-maximizing inputs for a constraint set. It supports a wide array of language features, as demonstrated by several frontends. The key distinctions are that Scimitar optimizes over any program variable including ones that are used across nested functions, and that solver invocations are nested within normal functional programs, which allows the virtual machine to direct control flow according to optimization results.

High Level Constructs within Solvers. MiniZinc [Nethercote et al. 2007] is a high-level, typed, mostly first-order, functional constraint optimization modeling language. It offers facilities for abstraction such as let-bindings and defining predicates and functions, but unlike Scimitar it cannot manipulate these as first class. MiniZinc also has a stronger compile-time/run-time distinction compared to Scimitar. MiniZinc compiles to the simpler FlatZinc format before being handed off to solvers. Scimitar iteratively unfolds a problem to handle an a priori unbounded number of variables, whereas MiniZinc is more suited to problems with a well specified fixed domain.

7 Future work

There are several potential directions for future work. One useful direction would be adding variable-length lists, whose encoding is more involved than the types we explored. Support for lists would allow us to provide higher-order list functions such as map, foldl, zip, and every, which could be used for elegant presentations for problems such as subset sum. Although not conceptually difficult, adding variable-length lists would require a large engineering effort.

Another valuable addition would be to progressively inline recursion to a greater depth, a feature that is available in some other solver tools. Although our encoding allows for efficiently solving for the minimum of value of some recursive expression, if the user does not have some knowledge of the upper bound, the initial problem might be infeasible. In such cases, it might make sense to retry with a higher limit, at the cost of longer compile and solve times.

Scimitar supports binary variable multiplication, but a similar encoding could enable general integer–integer and integer–float variable multiplication. This is limited by the bounds of the integer, as the number of constraints is directly proportional to the bounds. The encoding is practical and useful for multiplying when one variable's bounds are small.

For large bounds or float–float variable multiplication, it is not practical to use a MILP solver. Supporting this and other functions without a linear relaxation would require a non-linear approach such as using a Quadratically Constrained Quadratic Programming (*QCQP*) solver.

We believe there are several ways to improve Scimitar's compile times. These include streamlining generated constraints to eliminate redundancies and unnecessary complexity, incorporating *special ordered set* (SOS) constraints in place of certain binary variables, and amortizing multiple successive compilations of similar problems.

8 Conclusion

We described Scimitar, a language combining functional and symbolic reasoning compiled to optimization problems. These optimization problems are solved using an off-the-shelf MILP solver, and the solutions are used to determine program values and to direct control flow. Scimitar allows users to automatically take advantage of encoding techniques that must normally be written by hand, including solve-time conditionals, bounded inlining, and loop unrolling. We also presented a semantics for this language, showing how the host language interacts with the solver inside of a minimize o e term. We gave several examples of Scimitar in action, including benchmarks that demonstrate Scimitar's features. While currently Scimitar's compile time is slow, its quick solve performance supports the argument in favor of a dedicated MILP solver for optimization applications. In summary, Scimitar takes an important first step in demonstrating the potential benefits of an optimization-aided language.

Data-Availability Statement

Scimitar's implementation and benchmarks are available online [Bragg et al. 2024].

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments and Brian Lachance for his extensive Racket expertise. This research was supported in part by a Draper Scholarship.

A Appendix: Formalization of Scimitar's Source Language

A.1 Types

Figure 10 shows the Scimitar type system. Types provide guarantees about what the solver receives in the compiled code, and enable several important features (see Section 4).

Types τ range over a scalar interval $\mathcal{I}$ with a size or "measure" μ, the unit type, tuples $\tau \times \tau$, records $\langle x : \tau, \ldots \rangle$, and functions $\tau \to \tau$.

Tensors are given the type $\mathcal{I}^{\mu}$; vectors are a special case for $\mu = k$. While any closed interval $\mathcal{I}$ of the reals is a valid scalar type, the intervals of most interest to us are the reals $\mathbb{R}$ and the non-negatives $\mathbb{R}_{+}$, the natural numbers $\mathbb{Z}$, and the set $\{0, 1\}$, $\mathbb{I}$.

Types are built on the idea that we can model any value with a non-recursive algebraic type as a vector of type $\mathbb{R}^k$; other types are sugar on top of this basic type. The implementation flattens tensors, tuples, records, and primitives according to the dimensionality and range of their type.

$$\begin{array}{l}
\mathcal{I} ::= \mathbb{R} \mid \mathbb{R}_{+} \mid \mathbb{I} \mid \mathbb{Z} \\
\mu ::= k \mid k \times \mu \\
\tau ::= \mathcal{I}^{\mu} \mid () \mid \tau \times \tau \\
\quad \mid \quad \langle x : \tau, \ldots \rangle \mid \tau \to \tau \\
k \in \mathbb{Z}
\end{array}$$

Figure 10. The Scimitar types

A.2 Additional Semantics

In Section 3.1 discussed the interface between host and solver semantics for Scimitar. The intuition behind the functional solver semantics can be laid out using a complete trace-based semantics, as presented in Figures 11, 12, and 13. Because the whole program is evaluated at once, we can't analyze individual asserts in isolation to determine whether they will be violated. Even something as simple as `(= a b)` could for example make the program infeasible, force a solution, or have no effect. Solver semantics that evaluate to concrete values prove difficult to specify, as this is tantamount to presenting an entire solver algorithm, e.g. branch and cut. However, our trace-based approach can give us some insight as to what the solver sees before it begins its decision procedure.

The judgment form states that given some MILP problem that minimizes some objective o subject to constraints C, we can find an equivalent Scimitar program expression e that leads to the variable or value x. By "leads to," we mean specifically that building the MILP problem equivalent to the operation e requires the solver to be aware of x, which the problem may use in other constraints. It represents knowledge that the solver has about this expression.

Using these rules, we can explore informally how a user should expect a program to behave within a solve expression. The basic solver semantic concepts are variables and

$$\frac{\alpha, \beta \text{ are global constants}}{\min \alpha \cdot x + \beta; \top \vdash x \rightsquigarrow x}\ \textsc{Var} \qquad \frac{}{\min 0; \top \vdash v \rightsquigarrow v}\ \textsc{Val}$$

$$\frac{\begin{array}{c}\min o_g; C_g \vdash e_g \rightsquigarrow x_g \\ \min o_t; C_t \vdash e_t \rightsquigarrow x_t \qquad \min o_f; C_f \vdash e_f \rightsquigarrow x_f \\ x \text{ is fresh} \qquad C_x \equiv x = x_g * x_t + (1 - x_g) * x_f \\ C_{tg} \equiv x_g \Longrightarrow C_t \qquad C_{fg} \equiv (1 - x_g) \Longrightarrow C_f \\ C \equiv C_g \wedge C_{tg} \wedge C_{fg} \wedge C_x\end{array}}{\min o_g + o_t + o_f; C \vdash \mathsf{if}\ e_g\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \rightsquigarrow x}\ \text{ITE}$$

$$\frac{\begin{array}{c}\min o_i; C_i \vdash e_i \rightsquigarrow y_i \\ C_y \equiv y_i = \langle y_1, \ldots, y_n \rangle \qquad n < \text{unroll limit} \\ \min o_1; C_1 \vdash e_b[x \mapsto y_1] \rightsquigarrow x_1 \\ \vdots \\ \min o_n; C_n \vdash e_b[x \mapsto y_n] \rightsquigarrow x_n \\ C \equiv C_1 \wedge \ldots \wedge C_n \wedge C_i \wedge C_y\end{array}}{\min o_1 + \ldots + o_n + o_i; C \vdash \mathsf{for}\ x \Leftarrow e_i\ \mathsf{do}\ e_b \rightsquigarrow x_n}\ \textsc{For}$$

$$\frac{\begin{array}{c}\min o_i; C_i \vdash e_i \rightsquigarrow y_i \\ C_y \equiv y_i = \langle y_1, \ldots, y_n \rangle \qquad n < \text{unroll limit} \\ x_s \text{ is fresh} \qquad C_s \equiv x_s = x_1 + \ldots + x_n \\ \min o_1; C_1 \vdash e_b[x \mapsto y_1] \rightsquigarrow x_1 \\ \vdots \\ \min o_n; C_n \vdash e_b[x \mapsto y_n] \rightsquigarrow x_n \\ C \equiv C_1 \wedge \ldots \wedge C_n \wedge C_i \wedge C_y \wedge C_s\end{array}}{\min o_1 + \ldots + o_n + o_i; C \vdash \mathsf{sum}\ x \Leftarrow e_i\ \mathsf{of}\ e_b \rightsquigarrow x_s}\ \textsc{Sum}$$

Figure 11. Scimitar's functional solver semantics for variables, values, and loops. The judgment form used here is trace-based: $\min o; C \vdash e \rightsquigarrow x$

constraints, and our objective is to accumulate these to feed to the solver. The judgment form is read as follows. For an expression e on the right hand side of the turnstile, we introduce constraints C and a solver objective $\min o$ on the left side. Together, these lead to either a variable x or a value v, where x represents how the solver interprets the constraint(s) introduced by the expression.

Figure 11 gives the rules for variables, values, conditionals and loops. Var is the only case where an objective is introduced, an affine formula over the variable used in the statement of the problem. Values (Val) can only lead to themselves. Conditionals (ITE) use the objectives of all subexpressions, and the conditional behavior is represented in the constraints: first, we introduce a constraint C_x that governs what the overall expression leads to. This equation reads that depending on the binary variable x_g, either x_t or x_f are non-zero. In C_{tg} and C_{fg} we use the variable x_g to enable or disable the constraint sets C_t and C_f respectively. This is just an intuition—our actual implementation is more subtle. For

$$\frac{n \geq \text{inline/unroll limit}}{\phi; n; \text{PC} \implies \bot \vdash e \Downarrow e}\ \textsc{InlineFail}$$

$$\frac{}{\min 0; \top \vdash \lambda x\ .\ e \rightsquigarrow (\!|\lambda x\ .\ e|\!)}\ \textsc{Lambda}$$

$$\frac{\begin{array}{c}\phi = \{f \mapsto e_f\}\\ \phi; n+1; C \vdash e[f \mapsto e_f] \Downarrow e'\\ n < \text{inline/unroll limit}\end{array}}{\phi; n; C \vdash e \Downarrow e'}\ \textsc{Inline}$$

$$\frac{\begin{array}{c}\min o_1; C_1 \vdash e_1 \rightsquigarrow (\!|\lambda x_f\ .\ e_f|\!)\\ \min o_2; C_2 \vdash e_2 \rightsquigarrow x_2\\ \min o_f; C_f; \vdash e_f[x_f \mapsto x_2] \rightsquigarrow x\end{array}}{\min o_1 + o_2 + o_f; C_1 \wedge C_2 \wedge C_f \vdash e_1 e_2 \rightsquigarrow x}\ \textsc{Apply}$$

$$\frac{\begin{array}{c}\min o_f; C_f \vdash e_f \rightsquigarrow v_f\\ \phi = \{f \mapsto v_f\} \quad \phi; 0; C'_f \vdash v_f \Downarrow v'_f\\ \min o; C \vdash e[f \mapsto v'_f] \rightsquigarrow x \quad C' \equiv C_f \wedge C'_f \wedge C\end{array}}{\min o_f + o; C' \vdash \mathsf{letrec}\ f \Leftarrow e_f\ \mathsf{in}\ e \rightsquigarrow x}\ \textsc{Letrec}$$

Figure 12. Scimitar's functional solver semantics: functions

$$\frac{\min o_1; C_1 \vdash e_1 \rightsquigarrow x_1 \quad \ldots \quad \min o_n; C_n \vdash e_n \rightsquigarrow x_n}{\min o_1 + \ldots + o_n; C_1 \wedge \ldots \wedge C_n \vdash e_1; \ldots; e_n \rightsquigarrow x_n}\ \textsc{Seq}$$

$$\frac{\begin{array}{c}x \text{ is fresh}\\ \min o_1; C_1 \vdash e_1 \rightsquigarrow x_1 \qquad \min o_2; C_2 \vdash e_2 \rightsquigarrow x_2\end{array}}{\min o_1 + o_2; C_1 \wedge C_2 \wedge x_1 + x_2 = x \vdash e_1 + e_2 \rightsquigarrow x}\ \textsc{Plus}$$

$$\frac{\begin{array}{c}x \text{ is fresh}\\ \min o_1; C_1 \vdash e_1 \rightsquigarrow v_1 \qquad \min o_2; C_2 \vdash e_2 \rightsquigarrow x_2\end{array}}{\min o_1 + o_2; C_1 \wedge C_2 \wedge v_1 \cdot x_2 = x \vdash e_1 \cdot e_2 \rightsquigarrow x}\ \textsc{Times}$$

$$\frac{y \text{ is fresh} \quad \min o; C \vdash e \rightsquigarrow x \quad \min o_j; C_j \vdash i \rightsquigarrow j}{\min o + o_j; C \wedge C_j \wedge x_j = y \vdash e_i \rightsquigarrow y}\ \textsc{VecIx}$$

$$\frac{\min o_1; C_1 \vdash e_1 \rightsquigarrow x_1 \qquad \min o_2; C_2 \vdash e_2 \rightsquigarrow x_2}{\min o_1 + o_2; C_1 \wedge C_2 \wedge x_1 \preceq x_2 \vdash \mathsf{assert}\ e_1 \preceq e_2 \rightsquigarrow 0}\ \textsc{Ineq}$$

$$\frac{\begin{array}{c}x \text{ is fresh}\\ \min o_1; C_1 \vdash e_1 \rightsquigarrow x_1 \quad \ldots \quad \min o_n; C_n \vdash e_n \rightsquigarrow x_n\\ C \equiv C_1 \wedge \ldots \wedge C_n \wedge x = \langle x_1, \ldots, x_n \rangle\end{array}}{\min o_1 + \ldots + o_n; C \vdash \langle e_1, \ldots, e_n \rangle \rightsquigarrow x}\ \textsc{Tuple}$$

Figure 13. Scimitar's functional solver semantics: other ops

the exact encoding and its explanation, see Figure 6. We structure our constraints like this to allow backwards reasoning about x_g. For and Sum are relatively straightforward. Sometimes we can statically decide what the loop upper bound is and unroll the loop or sum completely. Otherwise the exact behavior depends on reaching the unroll limit; when this happens (discussed in Section 3.4), which we can decide during solve time, a conditional contradiction constraint is introduced, which makes the overall problem infeasible.

Figure 12 shows the semantics for function definition, anonymous functions, and function application. Like the unroll limit for loops, we make use of a dynamic inlining limit while inlining functions, which we depict using a special judgment form. Given some function definition ϕ, an iteration limit n, and a conditional contradiction constraint C, Inline recursively replaces uses of ϕ in the expression e, yielding the completely inlined expression e'. We stop inlining when the inline limit is reached (InlineFail), recognizing that from the solver's perspective this point is not reached due to recursive execution but when a certain constraint over the path condition PC is uniquely feasible. I.e., the solver decides that the variables representing the path condition can only take on the values corresponding to the situation where the inlining limit is reached. For simplicity we take the path condition for granted, and exclude its construction from these judgments.

Note that during function definition (Letrec) we pre-inline all functions up to this inlining limit, and then inline it into the body of the letrec. The expression e_f must be a function; this is verified during type inference. The Lambda rule evaluates to function values; no environment is included, as all variables and values have been pre-substituted. Note that functions only constrain the overall problem if invoked in an application. Otherwise, their constraints are dropped. The Apply rule checks that the value of e_1 is a function value[3] before substituting the actual into the body and applying it. The case where e_1 is not a function cannot occur, as is caught during type inference.

In Figure 13 we give the rest of the rules. Most of these are self-explanatory, but we want to draw attention to two: Times and Ineq. In multiplication, we allow only constant values or binary variables in one position or the other. In a constraint rule, after analyzing the subexpressions, we directly add this as a constraint without the expression leading to anything.

[3]We hand wave a little here—these semantics allow for the possibility that this is actually a variable, i.e., they do not guarantee a function value. Because of restructuring during solve expression processing, a function value is actually guaranteed here.

B Appendix: Formalization of $\mathcal{O}$

The optimization problem language is shown in Figure 14. With the exception of primitive declarations d it is a strict subset of the grammar presented in Figure 4. The exception is the primitive declaration $\mathcal{P}\ (x_1, \ldots, x_n)\ (r)\ (y_1, \ldots, y_m)\ s$, which introduces some primitive $\mathcal{P}$ with inputs x, result r and local variables y, each of some type τ. Note that all variables in a primitive must have type ascriptions, unlike in Scimitar's functional language. Although the $\mathcal{F}$ program was already type checked, we retain these types to assist in lowering to the solver format. A primitive's body is fundamentally a conjunction of constraints C. This language is an augmented version of a typical mathematical programming problem $\min c^T x \text{ s. t. } Ax \preceq b$ augmented with the explicit distinction between input, result, and local variables.

The semantics of $\mathcal{O}$ differ from $\mathcal{F}$ in several important ways. First, the looping constructs in $\mathcal{O}$ must have known finite bounds, and recursion is disallowed. Loops can not have undetermined ranges with system-wide upper bounds, as this capability is taken care of when compiling $\mathcal{F}$. Additionally, no expression can contain any constraints or loops.

Notably absent are lambdas, let bindings, and conditionals. These have been completely stripped while compiling $\mathcal{F}$. Lambdas are omitted from $\mathcal{O}$ because primitives in $\mathcal{O}$ must be named, which avoids the complexity of scope and closed over variables. Let bindings are undesirable for similar reasons. While conditionals might be useful at this level, we omit them to keep the design of $\mathcal{O}$ closer to a constraint-focused optimization problem representation.

Finally, we can't include solver blocks, as the solver is already $\mathcal{O}$'s target. Anyhow, Scimitar does not currently support nested solve expressions. If support were added, we would handle this feature in $\mathcal{F}$.

Of note, the dynamic vector indexing and McCormick envelope encodings are performed in $\mathcal{O}$, rather than directly in $\mathcal{L}$. This is a legacy aspect, and we intend to move these into $\mathcal{L}$.

Although we give $\mathcal{O}$ a concrete syntax, allowing users to code in it directly, the intention is for users to program exclusively in $\mathcal{L}$.

$$\begin{array}{rcl}
e & ::= & x \mid v \mid e\ e \mid (\ e\ , \ldots,\ e\ ) \\
 & \mid & e + e \mid e \cdot e \mid e_e \mid \mathsf{sum}\ x \Leftarrow e\ \mathsf{of}\ e \\
s & ::= & C \mid s\ ; \ldots ;\ s \\
 & \mid & \mathsf{for}\ x \Leftarrow e\ \mathsf{do}\ s \\
C & ::= & \mathsf{assert}\ e \preceq e \\
v & ::= & () \mid n \mid \alpha \mid \langle\ v\ , \ldots,\ v\ \rangle \mid \mathcal{P} \\
d & ::= & \mathcal{P}\ (\ x : \tau \ldots x : \tau\ )(\ r : \tau\ )(\ y : \tau \ldots y : \tau\ )\ s
\end{array}$$

$$x, r, y \in \text{variables}$$
$$\mathcal{P} \in \text{primitives}$$
$$n \in \mathbb{Z} \quad \alpha \in \mathbb{R}$$

Figure 14. The optimization problem language $\mathcal{O}$

References

Martín Abadi and Gordon D. Plotkin. 2020. A simple differentiable programming language. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–28. https://doi.org/10.1145/3371106

Atilim Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research* 18 (2018), 1–43. http://jmlr.org/papers/v18/17-468.html

Nikolaj Bjorner. 2022. Advanced topics. https://microsoft.github.io/z3guide/docs/optimization/advancedtopics/ Accessed: 2023-04-12.

Rastislav Bodík, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with angelic nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL ’10)*. Association for Computing Machinery, New York, NY, USA, 339–352. https://doi.org/10.1145/1706299.1706339

Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. 2017. Programming with a Differentiable Forth Interpreter. In *International Conference on Machine Learning*. 547–556. http://proceedings.mlr.press/v70/bosnjak17a.html ISSN: 2640-3498 Section: Machine Learning.

Nate F. F. Bragg, Jeffrey S. Foster, and Philip Zucker. 2024. Scimitar: Onward! 2024 Artifact. https://doi.org/10.5281/ZENODO.13625532

Benjamin Braun. 2012. Compiling computations to constraints for verified computation. *UT Austin Honors Thesis HR-12-10* (2012).

Steven Diamond and Stephen Boyd. 2016. CVXPY: A Python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research* 17, 1 (2016), 2909–2913. http://jmlr.org/papers/v17/15-408.html

John Dombrowski. 2018. McCormick envelopes. https://doi.org/10.21985/N29T8M

Iain Dunning, Joey Huchette, and Miles Lubin. 2017. JuMP: A modeling language for mathematical optimization. *SIAM review* 59, 2 (2017), 295–320. https://doi.org/10.1137/15M1020575

Jean-Christophe Filliâtre. 2003. *Why: a multi-language multi-prover verification tool.* Technical Report. Research Report 1366, LRI, Université Paris Sud. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.12.3200&q=A%20Certified%20Multi-prover%20Verification%20Condition%20Generator.

Robert Fourer, David M Gay, and Brian W Kernighan. 1987. *AMPL: A mathematical programming language.* AT & T Bell Laboratories Murray Hill, NJ.

GAMS Development Corporation. 2022. Generalized disjunctive programs (gdps). https://www.gams.com/latest/docs/UG_EMP_DisjunctiveProgramming.html

Gurobi Optimization, LLC. 2023. Gurobi Optimizer Reference Manual. https://www.gurobi.com

Laurent Hascoët and Valérie Pascual. 2013. The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification. *ACM Trans. Math. Softw.* 39, 3, Article 20 (may 2013), 43 pages. https://doi.org/10.1145/2450153.2450158

Joey Huchette and Juan Pablo Vielma. 2019. A combinatorial approach for small and strong formulations of disjunctive constraints. *Mathematics of Operations Research* 44, 3 (2019), 793–820. https://doi.org/10.1287/moor.2018.0946

Joey Huchette and Juan Pablo Vielma. 2022. Nonconvex piecewise linear functions: Advanced formulations and simple modeling tools. *Operations Research* (2022). https://doi.org/10.1287/opre.2019.1973

Oleg Kiselyov and Chung-chieh Shan. 2009. Embedded probabilistic programming. In *IFIP Working Conference on Domain-Specific Languages*. Springer, 360–384. https://doi.org/10.1007/978-3-642-03034-5_17

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay - Secure Two-Party Computation System. In *USENIX Security Symposium*.

Oleksandr Manzyuk. 2012. A simply typed λ-calculus of forward automatic differentiation. *Electronic Notes in Theoretical Computer Science* 286 (2012), 257–272. https://doi.org/10.1016/j.entcs.2012.08.017

Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. 2007. MiniZinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming–CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings 13*. Springer, 529–543. https://doi.org/10.1007/978-3-540-74970-7_38

Alex Ozdemir, Fraser Brown, and Riad S. Wahby. 2020. CirC: Compiler infrastructure for proof systems, software verification, and more. Cryptology ePrint Archive, Paper 2020/1586. https://eprint.iacr.org/2020/1586

P.M. Pardalos. 1988. Linear complementarity problems solvable by integer programming. *Optimization* 19, 4 (1988), 467–474. https://doi.org/10.1080/02331938808843365

Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 2 (2008), 1–36. https://doi.org/10.1145/1330017.1330018

Juan P Ruiz, Jan-H Jagla, Ignacio E Grossmann, Alex Meeraus, and Aldo Vecchietti. 2012. Generalized disjunctive programming: Solution strategies. In *Algebraic Modeling Systems*. Springer, 57–75. https://doi.org/10.1007/978-3-642-23592-4_4

The MathWorks, Inc. Copyright 2013-2020. *intlinprog*.

Emina Torlak and Rastislav Bodík. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software - Onward! ’13*. ACM Press, Indianapolis, Indiana, USA, 135–152. https://doi.org/10.1145/2509578.2509586

Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96. https://doi.org/10.1017/S1471068411000494

Wikipedia contributors. 2024. Big M method — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Big_M_method&oldid=1227030411 [Online; accessed 28-August-2024].

Received 2024-04-25; accepted 2024-08-08