

High-Level Optimization of Abstract Data Types

Anthony Hunt and Emil Sekerinski
McMaster University,
Hamilton, Canada



Programming With Abstract Data Types

- System specifications use sets and relations
 - Expressive and provable
 - Semantics from mathematics

Syntax	Label	Syntax	Label
$\text{set}(T)$	Unordered, unique collection	$S \cup T$	Union
$S \leftrightarrow T$	Relation, $\text{set}(S \times T)$	$S \cap T$	Intersection
$S \rightarrow \mathbb{N}$	Bag/Multiset	$S \setminus T$	Difference
$\mathbb{N} \rightarrow S$	Sequence	$S \times T$	Cartesian Product
$\{x, y, \dots\}$	Set Enumeration	$\text{dom}(R)$	Domain
$\{x \cdot P \mid E\}$	Set Comprehension	$R[S]$	Image

Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.
David Gries and Fred Schneider. *A Logical Approach to Discrete Math*. Springer New York, 1993.

Example - Visitor Information System

- Academics attend workshops throughout CASCON 2025
- Every workshop must be held in one room
- Visitors may attend at most one workshop at a time

$Visitor = \{J. Nelson, Mark, Ehsan\}$

$Room = \{1,2,3,4\}$

$Workshop = \{CDP, SENGEC, COGAI\}$

$location: Workshop \rightarrow Room$

$attends: Visitor \nrightarrow Workshop$

Total Injective Function

Partial Function

Then, the number of meals to prepare for a specific $room$ would be:

$card((location^{-1} \circ attends^{-1})[\{room\}])$

Relational Image

Example - Warehouse Inventory System

- Furniture material catalogue
- Warehouse inventory
- Product recipes

$Material = \{2 \times 4 Plank, Hex Bolt, 3 Inch Screw, \dots\}$

$Price = \mathbb{N}_0$

$Product = \{Cabinet, Desk, Bookshelf, \dots\}$

$catalogue: Material \rightarrow Price$

$inventory: bag[Material]$

$recipes: Product \rightarrow bag[Material]$

Equivalent to: $Material \nrightarrow \mathbb{N}$

Restocking price, with a target inventory $inventory_t$:

$$\sum p \mapsto n_m \cdot$$

$$p \mapsto n_m \in catalogue^{-1} \circ (inventory_t - inventory) \mid p \times n_m$$

Bag Difference

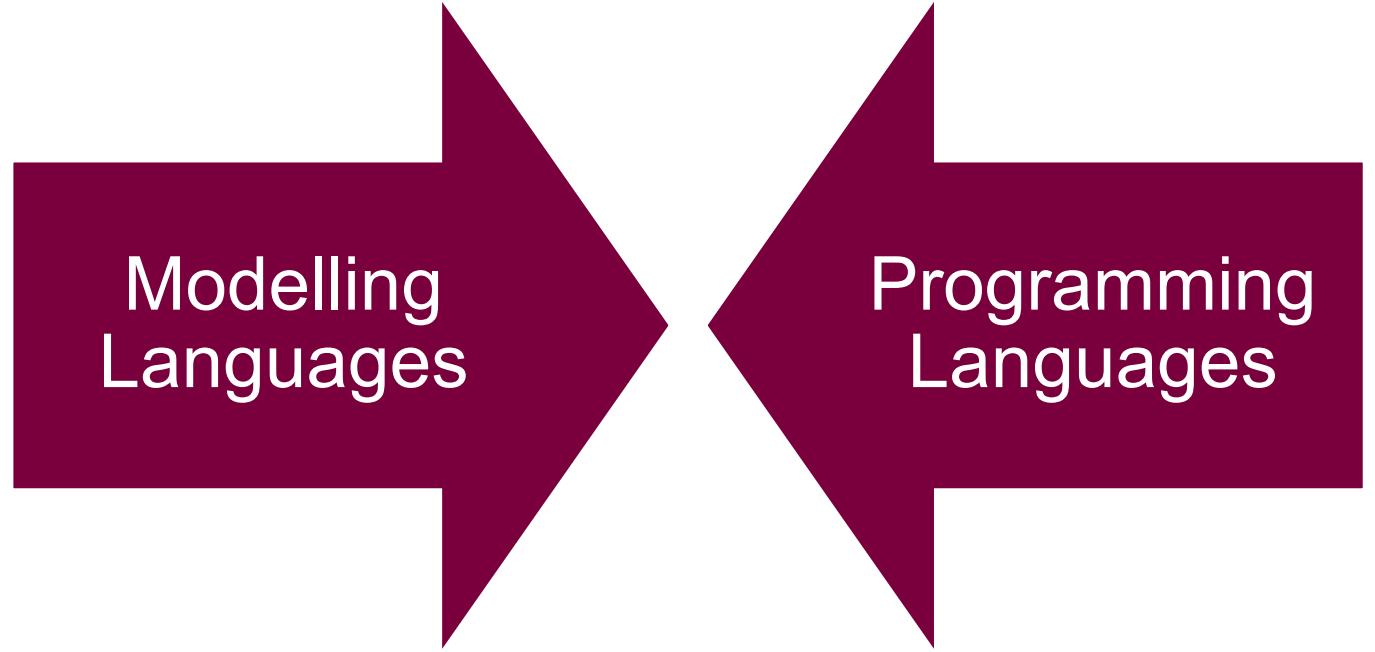
Abstract Data Types

In Programming and Modelling Languages

		Sets				Relations							Efficiently Executable	Implementation Free		
		U	n	x	{ $x \cdot P \mid E$ }	dom	<	U	n	\leftrightarrow	o	R[S]	R^{-1}	Multiplicity		
Programming Languages	Python	✓	✓	✓	✓	✓	X	X	*	✓	X	*	X	X	*	X
	Haskell	✓	✓	✓	*	✓	*	X	✓	✓	*	*	✓	*	✓	X
	Rust	✓	✓	*	*	✓	X	X	X	*	X	✓	X	X	✓	X
	C	*	*	*	X	*	*	*	*	*	*	*	*	X	✓	X
Modelling Languages	SetL	✓	✓	*	✓	✓	✓	✓	✓	✓	*	✓	*	*	*	*
	UML	✓	✓	✓	X	✓	X	X	X	✓	✓	X	✓	*	✓	
	Event-B	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	*	✓
	Alloy	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	✓

Goals

- Syntax close to discrete mathematics
- Simple, small, and usable
- High-level set-theory optimization
- Efficient in memory and time



Related Work

- ProB animation engine for Event-B [1]
- GHC rewrite rules [2]
- Rewrite systems for set-theory focused optimization [3,4,5]
- SETL data type *lowering* from abstract types to suitable concrete structures [6,7]

[1] Michael Leuschel. *Programming in B: Sets and Logic all the Way Down*. In: LPOP 2022.

[2] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. *Playing by the rules: rewriting as a practical optimisation technique in GHC*. In: 2001 Haskell Workshop. ACM SIGPLAN. Sept. 2001.

[3] Maximiliano Cristia and Gianfranco Rossi. *{log}: Programming and Automated Proof in Set Theory*. In: LPOP 2022.

[4] Elco Visser, Zine-el-Abidine Benissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In: ICFP 1998.

[5] Douglas R. Smith and Stephen J. Westfold. *Transformations for Generating Type Refinements*. In: FM 2019.

[6] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. 1979. *Automatic data structure selection in SETL*. In: POPL 1979.

[7] Stefan M. Freudenberger, Jacob T. Schwartz, and Micha Sharir. *Experience with the SETL Optimizer*. Association for Computing Machinery, 1983

Optimizing Abstract Data Type Operations (in Python)

$$(S \cup T) \cap V$$

```
# Python translation:  
ret = set()  
for x in S:  
    ret.add(x)  
# Step 1: len(ret) == len(S)  
for x in T:  
    ret.add(x)  
# Step 2: len(ret) <= len(S) + len(T)  
for x in ret:  
    if x not in V:  
        ret.remove(x)  
# Step 3: len(ret) <= len(V)
```



$$(S \cap V) \cup ((T \setminus S) \cap V)$$

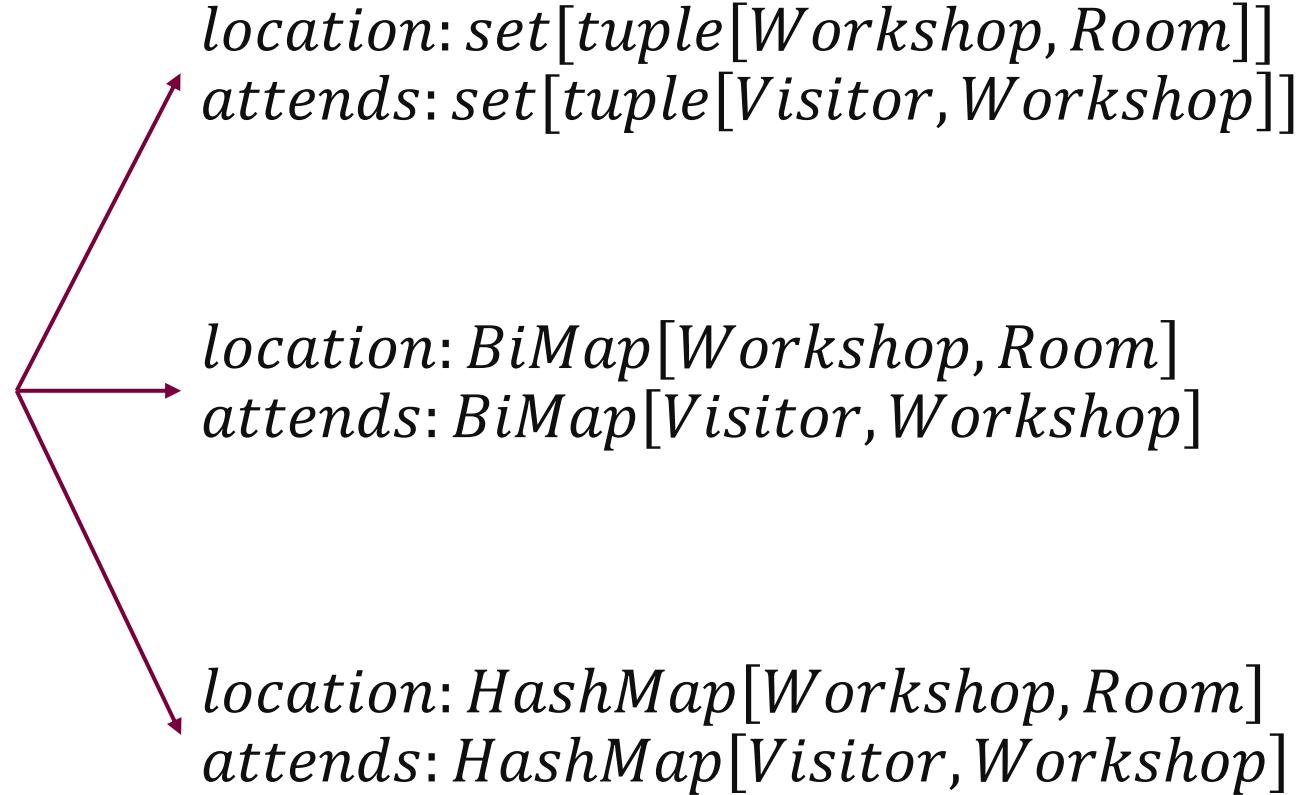
```
# New Python translation:  
ret = set()  
for x in S:  
    if x in V:  
        ret.add(x)  
# Step 1: len(ret) <= min(len(S), len(V))  
for x in T:  
    if x not in S and x in V:  
        ret.add(x)  
# Step 2: len(ret) <= min(len(S) + len(T), len(V))
```

What if $\text{len}(S) + \text{len}(T) > \text{len}(V)$?

ret never grows larger than needed

Concrete Data Types of the Visitor Information System

*location: Workshop \rightarrow Room
attends: Visitor \nrightarrow Workshop*



Naïve Implementation of the Visitor Information System

Concrete Types

location: set[tuple[Workshop, Room]]
attends: set[tuple[Visitor, Workshop]]

Imperative Interpretation

$\text{card}((\text{location}^{-1} \circ \text{attends}^{-1})[\{\text{room}\}])$



$l := \text{inverse}(\text{location})$
 $a := \text{inverse}(\text{attends})$
 $c := \text{compose}(l, a)$

...

Library

```
proc inverse(r):  
    r' := {}  
    for k, v ∈ r do  
        r' := r' ∪ {(v, k)}  
    return r'
```

Runtime: $O(n)$
Memory: up to $O(n)$

```
proc compose(r1, r2):  
    r' := {}  
    for k, v ∈ r1 do  
        for v', t ∈ r2 do  
            if v = v' then  
                r' := r' ∪ {(k, t)}  
    return r'
```

Runtime: $O(n^2)$
Memory: $O(n)$

Designing Optimizations

- Set theory semantics for optimizations
- Strong type system with refinements
- Rewrite system:
 - Simplify operations
 - Select concrete data representations from refined types
 - Generate efficient code for every concrete representation and operation sequence

Term Rewriting for Hash-based Sets and Relations

1. Comprehension Construction
2. Generator Selection
3. Loop Lowering
4. Relational Subtyping Loop Simplification
5. Loop Code Generation
6. Replace and Simplify

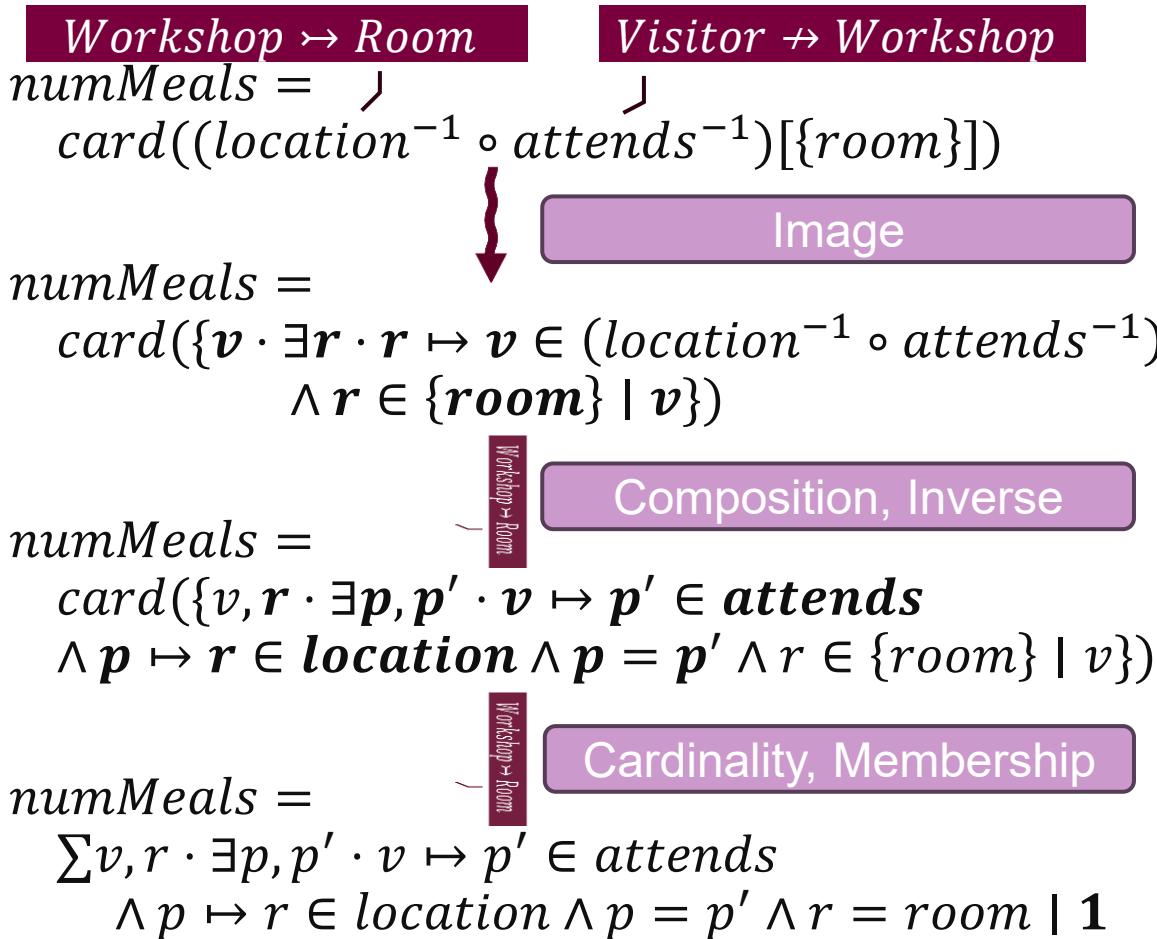
Phase 1: Set Comprehension Construction

Set-typed variables and literals are decomposed into set comprehensions

Rewrite Rules	
Predicate Operations	$S \cup T \rightsquigarrow \{x \cdot x \in S \vee x \in T \mid x\}$
	$S \cap T \rightsquigarrow \{x \cdot x \in S \wedge x \in T \mid x\}$
	$S \setminus T \rightsquigarrow \{x \cdot x \in S \wedge x \notin T \mid x\}$
Membership Collapse	$x \in \{E \mid P\} \rightsquigarrow \exists y \cdot P \wedge x = E$
Image	$R[S] \rightsquigarrow \{y \cdot \exists x \cdot x \mapsto y \in R \wedge x \in S \mid y\}$
Composition	$x \mapsto z \in R \circ Q \rightsquigarrow x, z \cdot \exists y, y' \cdot x \mapsto y \in R \wedge y \mapsto z \in Q \wedge y = y'$
Inverse	$x \mapsto y \in R^{-1} \rightsquigarrow y \mapsto x \in R$
Cardinality	$\text{card}(S) \rightsquigarrow \sum x \cdot x \in S \mid 1$

Post-condition:

- All set-like terms must be comprehensions



Phase 2: Generator Selection

Selecting element generators for use as iterables in generated for-loops

Rewrite Rules

Generator Selection

$$\bigwedge P_i \rightsquigarrow P_g \wedge \bigwedge_{P_i \neq P_g} P_{-i}$$

- P_g is of the form $x \in S$
- x is the quantifier's bound variable
- S is a set-like term

- In the case of multiple candidate generators, selection is based on heuristics

$$numMeals = \sum v, r \cdot \exists p, p' \cdot v \mapsto p' \in attends \\ \wedge p \mapsto r \in location \wedge p = p' \wedge r = room | 1$$



Generator Selection

$$numMeals = \sum \mathbf{v}, \mathbf{r} \cdot \exists p, p' \cdot \mathbf{v} \mapsto \mathbf{p}' \in \mathbf{attends} \\ \wedge \mathbf{p} \mapsto \mathbf{r} \in \mathbf{location} \wedge p = p' \wedge r = room | 1$$

Post-condition:

- All set-like terms must be comprehensions
- Quantifier predicates are in DNF
 - Guaranteed top-level-v operation
- One bound variable per generator
- All predicate v-clauses have an assigned generator

Phase 3: Loop Lowering

Start lowering expressions into imperative-like loops

Rewrite Rules			
Quantifier Generation	$\oplus E \mid P$	$a := \text{identity}(\oplus)$ $\rightsquigarrow \text{loop } P \text{ do}$ $a := a \oplus E$	
Chained Generators	$\text{loop } G_1 \wedge G_2 \text{ do}$ $a := a \oplus E$	$\rightsquigarrow \text{loop } G_1 \text{ do}$ $\text{loop } G_2 \text{ do}$ $a := a \oplus E$	
Disjunct Generators	$\text{loop } G_1 \vee G_2 \text{ do}$ $a := a \oplus E$	$\rightsquigarrow \text{loop } G_1 \text{ do}$ $a := a \oplus E$ $\text{loop } G_2 \wedge \neg G_1 \text{ do}$ $a := a \oplus E$	

Post-condition:

- No quantifiers exist within the AST
- No v -operators exist within a *loop*'s predicate
- All predicate v -clauses have an assigned generator

- \oplus is any of $\{\dots\}, \Sigma, \Pi, U, \cap$

$\text{numMeals} =$

$\sum_{v, r} \cdot \exists p', p \cdot v \mapsto p' \in \text{attends}$
 $\wedge p \mapsto r \in \text{location} \wedge r = \text{room} \wedge p = p' \mid 1)$

$\text{numMeals} := 0$

$\text{loop } v \mapsto p' \in \text{attends}$

$\wedge p \mapsto r \in \text{location} \wedge p = p' \wedge r = \text{room} \text{ do}$

$\text{numMeals} := \text{numMeals} + 1$

$\text{numMeals} := 0$

$\text{loop } v \mapsto p' \in \text{attends} \text{ do}$

$\text{loop } p \mapsto r \in \text{location} \wedge r = \text{room} \wedge p = p' \text{ do}$

$\text{numMeals} := \text{numMeals} + 1$

Quantifier Generation

Chained Generators

Phase 4: Relation Subtyping Loop Simplification

Eliminate unnecessary loops

Rewrite Rules

Restricted Generator

One-to-one Total Function

```

numMeals := 0
loop v → p' ∈ attends do
    loop p → r ∈ location ∧ r = room ∧ p = p' do
        numMeals := numMeals + 1
    
```

```

numMeals := 0
loop v  $\mapsto p' \in attends$  do
    if location(p') = room then
        numMeals := numMeals + 1
    end if
end loop

```

Post-condition:

- No quantifiers exist within the AST
 - No v-operators exist within a *loop*'s predicate
 - All predicate v-clauses have an assigned generator

Phase 5: Loop Code Generation

Lower into imperative *loop* structures

Rewrite Rules

Conjunct
Conditional

$$\text{loop } P_g \wedge \bigwedge P_i \text{ do } \rightsquigarrow \begin{array}{l} \text{if } \bigwedge_{free(P_i)} P_i \text{ then} \\ \quad \text{for } P_g \text{ do} \\ \quad \quad \text{if } \bigwedge_{\neg free(P_i)} P_i \text{ then} \\ \quad \quad \quad body \end{array}$$

Post-condition:

- Code is imperatively executable

$numMeals := 0$
loop $v \mapsto p' \in attends$ **do**
 if $location(p') = room$ **then**
 $numMeals := numMeals + 1$

{

$numMeals := 0$
for $v, p' \in attends$ **do**
 if $location(p') = room$ **then**
 $numMeals := numMeals + 1$

Visitor Information System - Translation

$location: Workshop \rightsquigarrow Room$
 $attends: Visitor \nrightarrow Workshop$

One-to-One Total Function

$$\begin{aligned} numMeals \\ = card((location^{-1} \circ attends^{-1})[\{room\}]) \end{aligned}$$



$$\begin{aligned} numMeals \\ = \sum_{v,r} \cdot \exists p, p' \cdot v \mapsto p' \in attends \\ \wedge p \mapsto r \in location \wedge p = p' \wedge r = room \mid 1 \end{aligned}$$



$$numMeals = \sum_v \cdot \exists p' \cdot v \mapsto p' \in attends \wedge location(p') = room \mid 1$$



```
numMeals := 0
for v, p' ∈ attends do
    if location(p') = room then
        numMeals := numMeals + 1
```

Warehouse Inventory System – Translation

catalogue: Material → Price **Many-to-One Total Function**

inventory: bag[Material] **Many-to-One Function**

recipes: Product → bag[Material]

Restocking price, with a target inventory $inventory_t$:

$$price = \sum p \mapsto n_m \cdot p \mapsto n_m \in catalogue^{-1} \circ (inventory_t - inventory) \mid p \times n_m$$

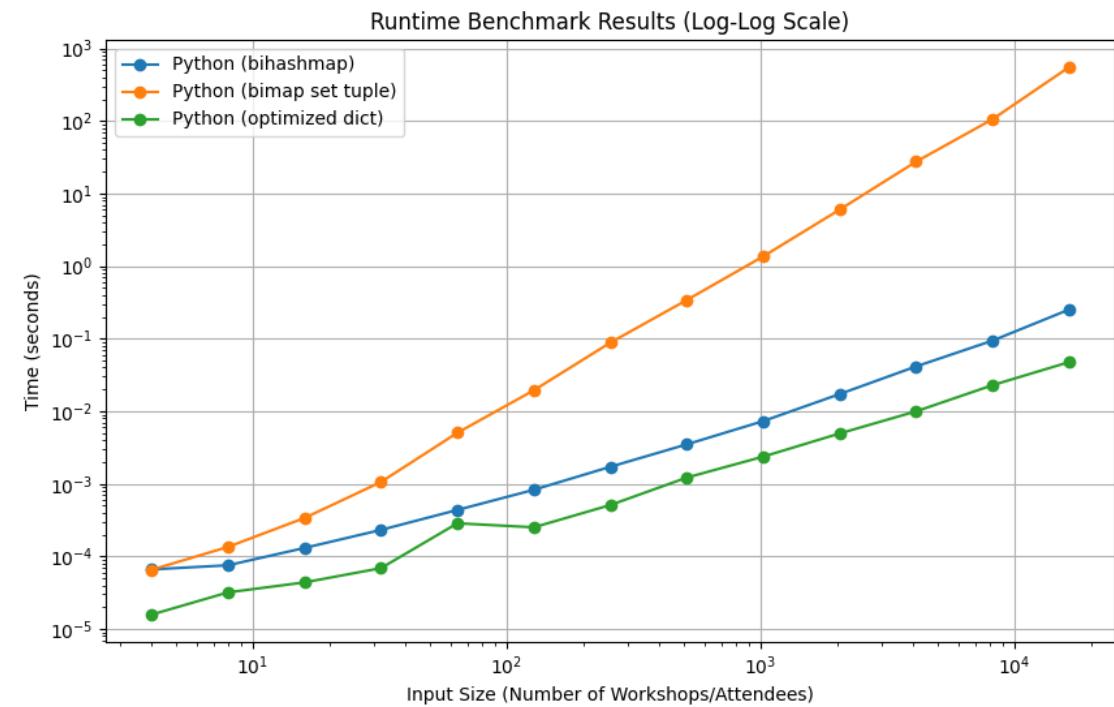
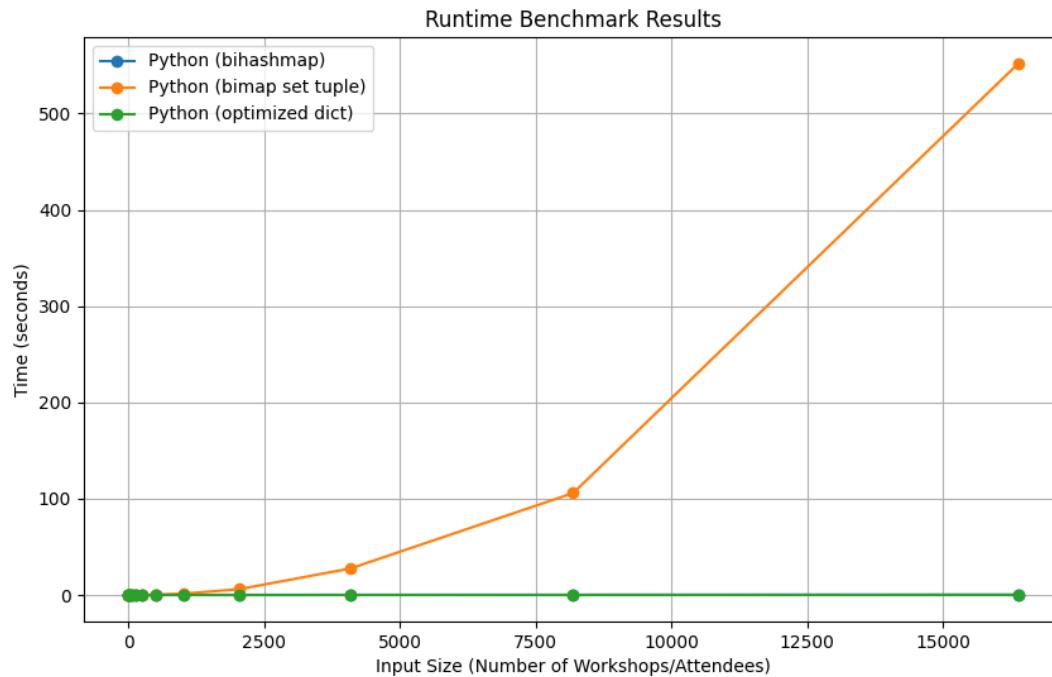
$$price = \sum m \mapsto n_m \cdot \exists n, m, p \cdot m \mapsto n_m \in inventory_t \wedge n = n_m - inventory(m) \wedge n \geq 0 \wedge p = catalogue(m) \mid p \times n_m$$

```
price := 0
for m, nm ∈ inventoryt do
    n := nm - inventory(m)
    if n ≥ 0 then
        price := price + catalogue(m) × n
```

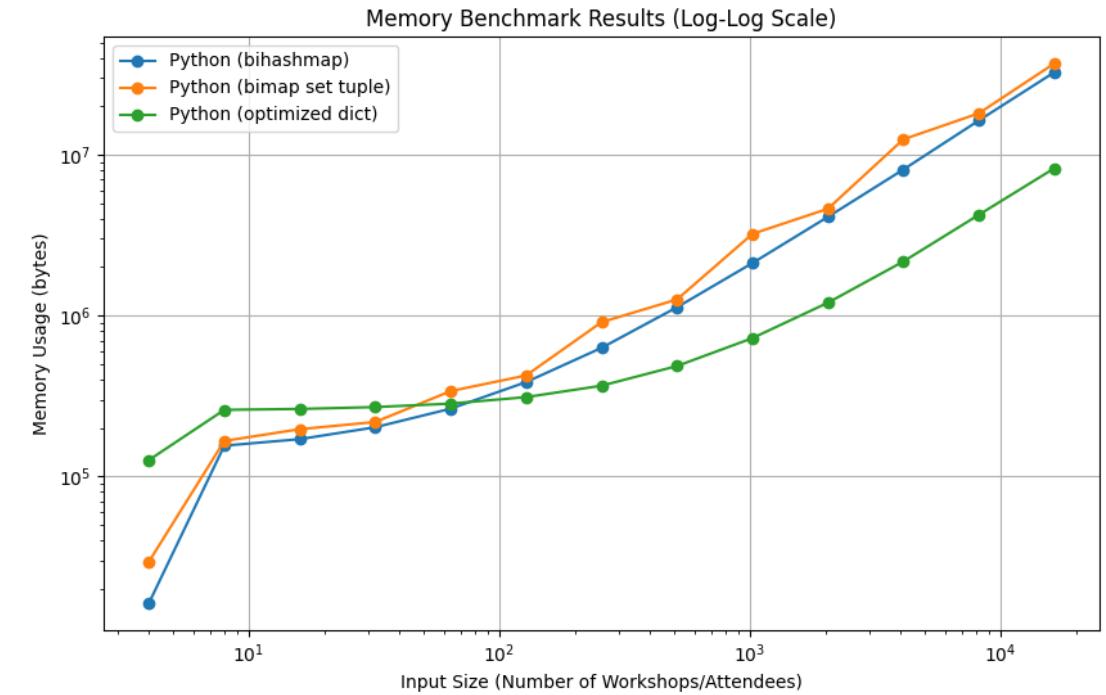
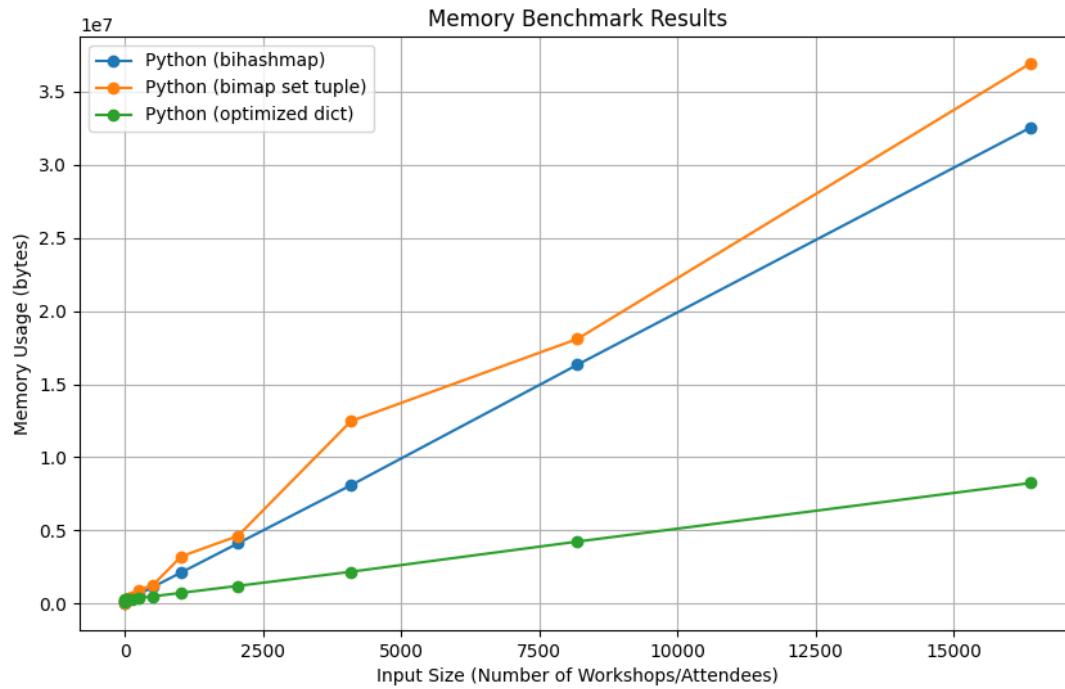
Preliminary Results

- Intermediate sets never grow larger than $\max(\text{starting sets}, \text{resulting set})$
- Unions, Relation Overriding, and Cartesian Products are limited in growth
- Other operators only decrease the size of a resulting set
- What about running time?...

Visitor Information System Benchmark – Runtime



Visitor Information System Benchmark – Memory Consumption



Examples Running Time

	Core Expression	Naïve Running Time	Optimized Running Time
Visitor Information System	$(location^{-1} \circ attends^{-1})[\{room\}]$	$O(location attends)$	$O(attends)$
Warehouse Inventory System	$\sum p \mapsto n_m \cdot p \mapsto n_m$ $\in (inventory_t - inventory) \circ catalogue^{-1}$ $ p \times n_m$	$O(inventory_t catalogue)$	$O(inventory_t)$

Operation Running Time With (Bi-directional) HashMaps

Operation	Expected Running Time
$S \cup T$	$O(S + T)$
$S \cap T$	$O(\min(S , T))$
$R[S]$	$O(\min(S , R))$
$R \circ Q$	$O(R Q)$
$(S \cup T) \cap U$	$O(\min(S + T , U))$
$S \cap R[T]$	$O(\min(S , R , T))$
$(R \circ Q)[S]$	$O(\min(S , R) + \min(R[S] , Q))$

- Simplify large groups of \wedge -predicates
- Eagerly apply conditions on sequential \vee -predicates

Current Compiler State and Roadmap

Development of a High-Level, Efficient, Set-Based Language

Model
Simulation
Specification

Optimizing
Rewrite Rules

Concurrency
Optimizations:
SIMD, Data
Parallelism

Data Type
Lowering

Code
Generation
(MLIR/LLVM)



References

- Michael Leuschel. *ProB2 Jupyter Notebooks GitLab*. 2020. https://gitlab.cs.uni-duesseldorf.de/general/stups/prob2-jupyter-notebooks-/blob/9d830112d9713d0cbc12d9d285f44ee61c827ccf/puzzles/Game_of_Life.ipynb
- Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.
- David Gries and Fred Schneider. *A Logical Approach to Discrete Math*. Springer New York, 1993.
- Robert Dewar. *The SetL Programming Language*. Bell Laboratories, 1979.
- Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. *Playing by the rules: rewriting as a practical optimisation technique in GHC*. In: 2001 Haskell Workshop. ACM SIGPLAN. Sept. 2001.
- Michael Leuschel. *Programming in B: Sets and Logic all the Way Down*. In: LPOP 2022.
- Maximiliano Cristia and Gianfranco Rossi. *{log}: Programming and Automated Proof in Set Theory*. In: LPOP 2022.
- Douglas R. Smith and Stephen J. Westfold. *Transformations for Generating Type Refinements*. In: Formal Methods. FM 2019 International Workshops. Springer International Publishing, 2020, pp. 371-387
- Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. 1979. *Automatic data structure selection in SETL*. In Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '79). Association for Computing Machinery, New York, NY, USA, 197–210. <https://doi.org/10.1145/567752.567771>
- Stefan M. Freudenberger, Jacob T. Schwartz, and Micha Sharir. 1983. *Experience with the SETL Optimizer*. ACM Trans. Program. Lang. Syst. 5, 1 (Jan. 1983), 26–45. <https://doi.org/10.1145/357195.357197>