# Optimizations in the Regular Games language

(Optymalizacje w języku Regular Games)

Jakub Cieśluk

Praca magisterska

**Promotor:** dr Marek Szykuła, mgr inż. Radosław Miernik

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

8 września 2025

**Abstract**

Optimizing compilers are the norm for modern general-purpose programming languages. They allow developers to write readable code, which is later optimized in terms of execution time or memory usage. This subject can also be of great use for domain-specific languages.

Regular Games is a language designed for General Game Playing, a field of artificial intelligence focused on developing agents that can play any arbitrary game, without knowing the rules beforehand. It was created at the University of Wrocław.

In this work, we present the optimization stage in the compiler for the Regular Games language. We describe the implemented transformations and data-flow analyses needed for them. We show how they influence game descriptions and provide results on how they impact the performance of the reasoner.

---

Kompilatory optymalizujące są standardem w nowoczesnych językach programowania ogólnego przeznaczenia. Pozwalają one programistom pisać czytelny kod, który jest następnie optymalizowany pod względem czasu wykonania lub zużycia pamięci. Ich użycie może być również bardzo przydatne w przypadku języków domenowych.

Regular Games to język zaprojektowany z myślą o General Game Playing, dziedzinie sztucznej inteligencji skupiającej się na tworzeniu agentów komputerowych, którzy mogą grać w dowolną grę, nie znając wcześniej jej zasad. Został on stworzony na Uniwersytecie Wrocławskim.

W niniejszej pracy przedstawiamy etap optymalizacji w kompilatorze języka Regular Games. Opisujemy zaimplementowane transformacje i analizy przepływu danych niezbędne do ich realizacji. Pokazujemy, jak wpływają one na opisy gier i przedstawiamy wyniki dotyczące ich wpływu na wydajność silnika wnioskowania.

# Contents

# Chapter 1

# Introduction

General Game Playing [1] (GGP) is a field of artificial intelligence aimed at creating programs that can play more than one game. Originally, game-playing agents were written specifically to be able to play one game, and their skills were not transferable to other games. They used different heuristic approaches and unique algorithms related to the specific game. Moreover, the creators of such agents had to know the rules of the game well.

In GGP, programs can play any arbitrary game without the need to know the rules beforehand. To achieve this, we need a formal description of the game rules, called a game description language. It typically covers a family of games in a human- and computer-readable way. To play a game, an agent is provided with a set of rules in this language or with a forward model (also called *reasoner*), which allows for simulating gameplay. This approach gained a lot of attention in the field of Reinforcement Learning because it provides a great place for experiments and performance tests. In this case, the simulation's efficiency plays a major role.

## 1.1 General Game Playing

The language that started the idea of GGP was Game Description Language [2]. It was expressive and covered a wide range of games, but at the same time was based on an inefficient system, which used logic resolution in the reasoner. The successor of GDL, called GDL–II [3], added support for randomness and incomplete information in games, but this came with a further regression in performance. With the founding of the GGP field, the annual International General Game Playing Competition began [4]. Newly designed languages aim to be faster, more expressive, and easier to use.

Ludii [5] is a language created to cover all traditional board games. It encodes high-level game features, called *ludemes*, which can be used to describe complex game rules cleanly and shortly. On the other hand, it makes the language very complicated, as it has over a thousand keywords defining various game concepts. Ludii has a big

database of available games, and is many times faster than the fastest reasoners for GDL. As a downside, it can be used only on the Ludii platform, which is written in Java and is source-available but not open source.

Regular Boardgames [6] (RBG) is the fastest available GGP language. It uses regular expressions for encoding games with the help of a simple macro system. The scope is board games, hence RBG includes a built-in concept of a board. It has a minimal syntax and only covers games with perfect information and deterministic rules. Primarily designed for achieving high efficiency, the rules in RBG are compiled to a reasoner in C++, which is highly optimized. At the same time, more complex games suffer from a long compilation time, mainly caused by very long game descriptions for complex games after macro unrolling.

## 1.2 Regular Games

Regular Games (RG), created at the University of Wrocław, is a successor of RBG. Instead of focusing on board games, it generalizes its predecessor concepts, removing the built-in constructs like board or arithmetic. Furthermore, it is universal for all turn-based games and handles imperfect information and randomness. Like RBG, it is equipped with a compiler that compiles game descriptions to a C++ module. While RBG used regular expressions for representing rules, RG shifts to a nondeterministic finite automaton (NFA), which is more succinct from a theoretical point of view [7] and more flexible. However, the drawback is that defining an NFA is very verbose and laborious for writing by a human developer. On the other hand, RG syntax is very minimal, allowing for easy processing and analysis.

High-level Regular Games (HRG) is a second language in the RG ecosystem. Its syntax resembles that of a general-purpose programming language, using constructs like loops and conditional expressions, which makes writing games more natural for developers. Games in HRG automatically translate to RG. In this way, RG serves only as the target language; games do not need ever be encoded directly in RG. On top of HRG can also exist languages more specific to different kinds of games, e.g., games with cards or with a dice. An example of such languages is *LineGames*, which simplifies writing variants of games similar to *Alquerque*.

RG ecosystem, presented in Figure 1.1, contains an Integrated Development Environment (IDE) [8], an automaton visualizer, and benchmarking tools. RG compiler includes a sophisticated optimization phase, which can highly improve the performance of the resulting reasoner module. The creators of RG also developed a mechanism of automatic translation of games written in RBG and GDL to RG (currently experimental), allowing all of them to reuse the same optimization pipeline and tooling.

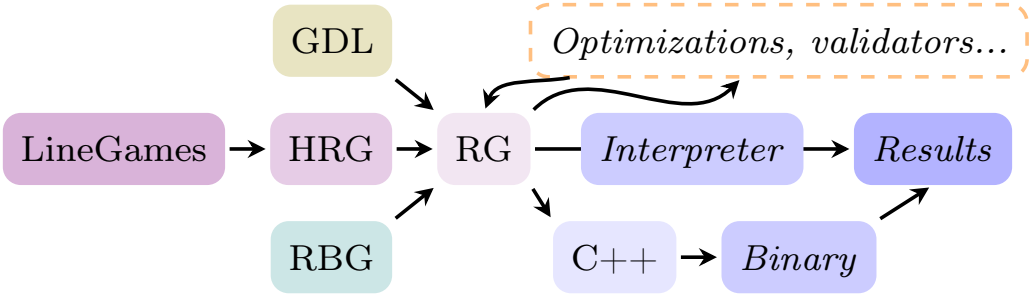


Figure 1.1: The Regular Games ecosystem. This work focuses on the *Optimizations* stage.

## 1.3 Optimizing pipeline

As mentioned before, a reasoner's performance plays a major role when deciding on a game description language. Both Ludii and RBG put focus on optimizing generated forward models, in Ludii by implementing single ludemes as efficiently as possible, while RBG tries to infer the most optimal C++ code by combining patterns found in the game rules. The sole generation of the forward model module in RG uses some ideas from its predecessor, but the optimization process starts much earlier.

Regular Games uses the approach of general-purpose programming languages and implements optimizations directly on the Abstract Syntax Tree (AST) [9, 10]. This way, we can detect inefficiencies in both small code blocks (*peephole optimizations* [11, 12]) and in the whole program. Most languages utilize a wide set of optimizations, where most of them are common between compilers [13, 14]. These transformations must be safe, i.e., they cannot change the semantics of the program [15].

In this thesis, we present the optimization stage for the RG language.

# Chapter 2

# The Regular Games language

This section describing the RG language is taken from a research paper prepared as a joint effort by Radosław Miernik, Marek Szykuła, Jakub Kowalski, Jakub Cieśluk, Łukasz Galas, and Wojciech Pawlik, currently submitted to the conference AAAI 26 [16].

A program written in Regular Games is called a *regular game*. It consists of types, constants, variables, pragmas, and an automaton in the form of a list of edges. These elements can appear in any order.

## 2.1 Types and values

A type in RG is either a set type or an arrow type. A domain of a set type is a finite set of symbols, denoted as `{ s1, s2, s3}`. An arrow type, denoted as `T1→T2`, has a key type `T1` and a value type `T2`. The domain of an arrow type consists of maps. Only set types can be used as key types. Type `T` is *assignable* to `U` if they are set types and have at least one common element, or if they are arrow types, and both their key types and value types are assignable.

A value in RG is either a symbol, denoted as `s`, or a map, which consists of a set of entries. A map entry is a key-value pair, denoted as `k: v`, or a default entry, denoted as `:v`, where `k` is a symbol and `v` is a value. Each map must have exactly one default entry.

## 2.2 Constants, variables, and type aliases

Constants describe immutable values. A constant, denoted as `const c: T = v;`, has a name `c`, a type `T`, and a value `v`. This value can reference other previously defined constants.

```
const empty: Piece = e;
const incr: Coord → Coord = {0:1, :2};
const pieceOfPlayer: Player → Piece = {X:X, :O};
const opponent: PlayerToPlayer = {X:O, :X};
```

Figure 2.1: Constants.

Variables represent mutable values. A variable, denoted as `var x: T = v;`, has a name `x`, a type `T`, and an initial value `v`, which can reference a constant or a previously defined variable.

```
var player: Player = X;
var board: Board = {:{:empty}};
var posX: Coord = 0;
```

Figure 2.2: Variables.

Type aliases, denoted as `type A = T;`, can be used instead of the type `T` they refer to. A type alias can reference other type aliases, but recursion is forbidden.

```
type Player = {X,O};
type PlayerToPlayer = Player → Player;
type Coord = {0,1,2};
type Piece = {X,O,e};
type Booard = Coord → Coord → Piece;
```

Figure 2.3: Type aliases.

## 2.3 Expressions

Given a state of all variables (mapping from their name to a value), an expression evaluates to a value of some type. There are three kinds of expressions:

- a **reference**, denoted by an identifier. It refers to a variable, a constant, or a symbol. Its type is the type of the referenced value. When evaluating, we first try to resolve it as a constant, then as a variable. If both attempts fail, we treat the identifier as a symbol.
- a **cast**, denoted as `T(e)`, where `T` is a type and `e` is an expression. The type of a cast expression is always `T` and its value is the value of `e`. We can cast an expression `e` of type $\texttt{T}_e$ to a type T only if $\texttt{T}_e$ is assignable to `T`.

- an **access**, denoted as $e_2[e_1]$, that consists of two subexpressions $e_1$ (called accessor), and $e_2$. Value $v$ of $e_2$ should be a map of an arrow type $T_1 \to T_2$, and value $k$ of $e_1$ should be a symbol of type $T_k$, where $T_k$ is assignable to $T_1$. The type of the result of the whole expression is $T_2$ The evaluated value of this expression is the value assigned to $k$ in map $v$, or the default value in this map, if there is no entry associated with key $k$.

```
board                          // Board
board[posX]                    // Coord → Piece
board[incr[posX][posX]]        // Piece
Player(X)                      // Player
pieceOfPlayer[Player(X)]       // Piece
```

Figure 2.4: Expressions.

## 2.4 Edges

Edge is denoted as $n_1$, $n_2$: <label>;. It consists of two nodes: a source node $n_1$ and a target node $n_2$, and a label. A node is denoted by its name. Incoming edges of a node $n$ are all edges in the game where $n$ is a target node. Similarly, outgoing edges of a node $n$ are all edges in the game where $n$ is a source node.

An automaton is a connected component in the graph describing the game. A game can consist of multiple connected components, but each game must contain a main automaton, which includes two special nodes: `begin` and `end`. Other automata can be accessed only through reachability checks (but we can also perform a reachability check on a main automaton), and thus are called reachability automata, or subautomata.

A path is a sequence of edges ($n_1$, $n_2$: <$l_1$>; $n_2$, $n_3$: <$l_2$>; ... ), where each edge starts in a node where the previous one ends.

A simple path is a path where only the first node has more than one incoming edge and only the last node has more than one outgoing edge.

Labels describe actions that can change the state of the game or determine if an edge is *legal* - it can be traversed. A path is legal if every edge on it is legal. A label is *valid* if it applies to rules defined below. Every label in a correct game description must be valid.

```
a, b: ;                    // skip label
a, b: player == X;         // comparison
a, b: board[0][0] != e;    // negated comparison
a, b: player = X;          // assignment to player
a, b: board[posX][0] = e;  // assignment to a map entry
a, b: posX = Coord(*);     // any assignment to posX
a, b: ? p → q;             // reachability check
a, b: ! p → q;             // negated reachability check
a, b: $ L;                 // tag L
a, b: $$ posX;             // variable tag posX
```

Figure 2.5: Labels.

A label is one of:

- a **skip label**, representing an empty action. Edges with skip labels are called *skip edges*. It is always valid and legal.
- a **comparison** $e_1$ == $e_2$. This action is legal if the comparison holds. The comparison can be negated $e_1$ != $e_2$, which is legal when the values of $e_1$ and $e_2$ are different. It is valid only if the type of $e_1$ is assignable to the type of $e_2$.
- an **assignment** $e_1$ = $e_2$, where $e_1$ is mutating the value under $e_1$. This label is always legal. For it to be valid, type $T_2$ of $e_2$ must be assignable to type $T_1$ of $e_1$.
- a **reachability check** ? p →q which can also be negated ! p →q. Here p and q are called *reachability targets*, where p is the *reachability start* and q is the *reachability end*. The reachability check is used to verify a complex condition. This action is legal if node q is reachable (or unreachable, if the check is negated) from p. Reachability targets can be a part of the main automaton or a reachability subautomaton. Reachability checks are not recursive, i.e., if a, b: ? p →q is a reachability check, then a must not be reachable from p. Note that all changes to the state of the game inside the reachability check are reverted after the check returns. Because of this, they can have a significant impact on the game's efficiency, since an assignment inside a check requires us to copy the game state.
- a **tag** $ t, this action is always legal and valid, and is used to distinguish player moves.

The game description also includes various pragmas, which serve as a hint to optimize runtime. We will not focus on them in this work.

## 2.5 Syntax sugar

There is some available syntax sugar, mostly to avoid listing every possible option or every member of some set type. A label *any assignment* is denoted as `e1 = T(*)`. An edge `n1, n2: x = T(*);` is equivalent to a set of parallel edges with an assignment to `x`, one for every member of type `T`, i.e., `n1, n20: x = 0; n1, n21: x = 1; ...` . Similarly, a *variable tag* label, denoted as `$$ v`, where `v` is a variable of a set type, is a shorthand for a tag with the value of variable `v`. It expands to a set a paths of length two, in the form of `n1, n20: v == 0; n20, n30: $0;`, for every member of type `T`.

## 2.6 Special definitions

Every game, by default, includes a few built-in definitions. Nodes `begin` and `end` specify automaton initial and final states. Once the final state is visited, the game ends. `keeper` and `random` are two special players, where the first manages the game and the second handles nondeterminism. Set type `Player` defines the players of the game, excluding the two mentioned beforehand. There are a few other basic types, including:

- `type Bool={0, 1}`.
- `type PlayerOrSystem` - `Player` type extended with `keeper` and `random` players.
- `type Score` - possible outcomes of the game, for each player, e.g., `type Score = {0, 1}`.
- `type Goals = Player→Score` - together with variable `var goals: Goals` keeps the score of every player.
- `type Visibility=Players→Bool` - using this type and the variable `var visibility: Visibility`, we specify if the current move is visible to all players. By default, moves are always visible to everyone.

## 2.7 Game description in Regular Games

In Figures 2.6 and 2.7, we can see the tic-tac-toe game described in RG and HRG.

```
type Player = {X,O};
type Score = {0,50,100};
type Coord = {0,1,2};
type Piece = {e,X,O};
type ColumnOfBoard = Coord -> Piece;
type Board = Coord -> ColumnOfBoard;
type PlayerToPlayer = Player -> Player;

const opponent: PlayerToPlayer = {X:O, :X};
const otherInLine1: Coord -> Coord = {0:1, :0};
const otherInLine2: Coord -> Coord = {:2, 2:1};
const onLRDiagonal: Coord -> Coord -> Bool = {0:{0:1,:0}, 1:{1:1,:0}, 2:{2:1,:0}, :{:0}};
const onRLDiagonal: Coord -> Coord -> Bool = {2:{0:1,:0}, 1:{1:1,:0}, 0:{2:1,:0}, :{:0}};

var goals: Goals = {:50};
var playerTurn: Player = X;
var posX: Coord = 0;
var posY: Coord = 0;
var board: Board = {:{:e}};

begin,turn: playerTurn = Player(X);

checkForEmpty,checkForEmptyX: posX = Coord(*);
checkForEmptyX,checkForEmptyY: posY = Coord(*);
checkForEmptyY,emptyExists: board[posX][posY] == Piece(e);

turn,move: ? checkForEmpty -> emptyExists;
turn,preend: ! checkForEmpty -> emptyExists;
preend,end: player = PlayerOrSystem(keeper);

move,chooseX: player = PlayerOrSystem(playerTurn);
chooseX,choosenX: posX = Coord(*);
choosenX,chooseY: $$ posX;
chooseY,choosenY: posY = Coord(*);
choosenY,check: $$ posY;
check,set: board[posX][posY] == Piece(e);
set,endmove: board[posX][posY] = Piece(playerTurn);

endmove,checkwin: player = PlayerOrSystem(keeper);
checkwin,win: ? checkline -> endcheckline;
checkwin,nextturn: ! checkline -> endcheckline;
nextturn,turn: playerTurn = opponent[playerTurn];

checkline,checklineH1:;
checklineH1,checklineH2: board[otherInLine1[posX]][posY] == Piece(playerTurn);
checklineH2,endcheckline: board[otherInLine2[posX]][posY] == Piece(playerTurn);
checkline,checklineV1:;
checklineV1,checklineV2: board[posX][otherInLine1[posY]] == Piece(playerTurn);
checklineV2,endcheckline: board[posX][otherInLine2[posY]] == Piece(playerTurn);
checkline,checklineLR1: onLRDiagonal[posX][posY] == 1;
checklineLR1,checklineLR2: board[otherInLine1[posX]][otherInLine1[posY]] == Piece(playerTurn);
checklineLR2,endcheckline: board[otherInLine2[posX]][otherInLine2[posY]] == Piece(playerTurn);
checkline,checklineRL1: onRLDiagonal[posX][posY] == 1;
checklineRL1,checklineRL2: board[otherInLine1[posX]][otherInLine2[posY]] == Piece(playerTurn);
checklineRL2,endcheckline: board[otherInLine2[posX]][otherInLine1[posY]] == Piece(playerTurn);

win,win1: goals[playerTurn] = Score(100);
win1,win2: goals[opponent[playerTurn]] = Score(0);
win2,end: player = PlayerOrSystem(keeper);

```

Figure 2.6: Tic-tac-toe in (low-level) Regular Games.

```
domain Piece = empty | x | o
domain Player = x | o
domain Position = P(I, J) where I in 0..2, J in 0..2
domain Score = 50 | 0 | 100

board : Position -> Piece = { P(I, J) = empty where I in 0..2, J in 0..2 }
position : Position

next_d1 : Position -> Position
next_d1(P(I, J)) = if I == J then P((I + 1) % 3, (J + 1) % 3) else P(I, J)
next_d2 : Position -> Position
next_d2(P(I, J)) = if I + J == 2 then P((I + 1) % 3, (J - 1) % 3) else P(I, J)
next_h : Position -> Position
next_h(P(I, J)) = P(I, (J + 1) % 3)
next_v : Position -> Position
next_v(P(I, J)) = P((I + 1) % 3, J)
op : Player -> Player
op(x) = o
op(o) = x

reusable graph existsNonempty() {
  position = Position(*)
  check(board[position] == empty)
}

reusable graph win() {
  branch {
    check(position != next_d1(position))
    check(board[position] == board[next_d1(position)])
    check(board[position] == board[next_d1(next_d1(position))])
  } or {
    check(position != next_d2(position))
    check(board[position] == board[next_d2(position)])
    check(board[position] == board[next_d2(next_d2(position))])
  } or {
    check(board[position] == board[next_h(position)])
    check(board[position] == board[next_h(next_h(position))])
  } or {
    check(board[position] == board[next_v(position)])
    check(board[position] == board[next_v(next_v(position))])
  }
}

graph turn(me: Player) {
  player = me
  position = Position(*)
  check(board[position] == empty)
  board[position] = me
  $$ position
  player = keeper
  if reachable(win()) {
    goals[me] = 100
    goals[op[me]] = 0
    end()
  }
  if not(reachable(existsNonempty())) {
    end()
  }
}

graph rules() {
  loop {
    turn(x)
    turn(o)
  }
}

```

Figure 2.7: Tic-tac-toe in High-Level Regular Games.

# Chapter 3

# Compilation process

The Regular Games compilation process consists of multiple steps. As input, it takes a file with a game description, in one of the supported languages. Currently, these include Regular Boardgames (RBG), High-level Regular Games (HRG), Game Description Language (GDL), and (low-level) Regular Games itself. The process starts by parsing the input into an appropriate Abstract Syntax Tree (AST), different for each language. If the code was not written in RG, the AST is translated into the Regular Games AST. Translation algorithms are not part of this work, but games coming from other languages utilize the same optimization pipeline. Then, we append built-in constructs to the game if any are missing.

## 3.1 Verification and optimization stage

Next comes the verification phase, which finds early incorrect game descriptions. In addition to type-checking, it performs multiple checks, ensuring:

- constants cannot be mutated;
- there should be no duplicated names of constants, variables, and type aliases;
- every map must have a default value, and there should be no duplicate keys;
- reachability checks should not be recursive;
- there should be no tags at self-loops (edges where source and target nodes are equal);
- variables used in variable tags must be of a set type.

At this point, the optimization phase starts. It is the most time-consuming part of the compilation process. A series of optimizing transformations runs in a fixed-point loop. Whenever one of them changes the description, the loop restarts. The

verification phase is rerun after each restart. Thanks to this, we know if every intermediate state of the game description is valid. Every transformation can be toggled on separately, which allows for compromises between higher reasoner's efficiency and shorter compilation time.

After we have reached a fixed point, the compiler calculates pragmas, which serve as runtime hints. Finally, the optimized AST with added pragmas is sent to a separate process, which generates code for a forward model in C++. This process is not a part of this work. A forward model is a set of functionalities for managing game play – creating an initial state, calculating all legal moves at each state, applying a move, determining if a state is final, and deciding the game's outcome.

## 3.2 Data–flow analysis

When some optimizations need knowledge only about a local part of the automaton, others require a more global view of the whole game. This view can include knowledge of whether a variable has a known value at a given node, or which definition reaches a given variable reference. To gain such information about the code, we use *data-flow analysis* [17, 18]. It is a process that collects data about the program at every step of the execution. Depending on the analysis, the data might differ, but all of them use a unified framework. It is based on passing the knowledge along the control flow [19] and changing it based on encountered statements and expressions.

In the Regular Games compiler, the analysis uses an iterative worklist algorithm. The *Domain* type represents the result knowledge at each node. Functions *bot* and *extreme* define, respectively, the default knowledge at an unvisited node and knowledge at the game's entry point (`begin` node). For changing the knowledge when traversing an edge, we use the function *transfer*, or a pair of functions *gen* and *kill*. Multiple incoming edges can meet at a node. In such cases, the *join* function defines how to merge knowledge coming from them.

```
pub trait Analysis {
  type Domain: Clone + PartialEq;
  fn bot() -> Domain;
  fn extreme(program: &Game<Id>) -> Domain;
  fn gen(input: Domain, edge: Edge<Id>) -> Domain
  fn join(a: Domain, b: Domain) -> Domain;
  fn kill(input: Domain, edge: Edge<Id>) -> Domain
  fn transfer(input: Domain, edge: Edge<Id>) -> Domain {
    gen(kill(input, edge), edge)
  }
}
```

The main part of the task is done in the *Worker* struct. It takes some analysis and a *Flow*, and runs the analysis on a game. The mutable *result* field stores the

knowledge at each node of the game.

*Flow* describes the data flow of the game. It can be either forward (along the edges) or backward (against the edges). Depending on the analysis, we might want to exclude reachability subautomata. Currently, only forward flow is used in the implemented analyses.

The *Worker* uses an iterative worklist algorithm. At the beginning, the worklist is initialized with all game nodes. We transfer knowledge through the nodes until the worklist is emptied. After each change to the knowledge at some node, we add its successors (in the *Flow*) to the worklist to propagate the update. Transferring the knowledge means taking all edges coming into a node, first transferring knowledge through the edge, and then merging the knowledge from them.

```
struct Worker<A: Analysis> {
  analysis: &A,
  flow: &Flow,
  result: BTreeMap<Node<Id>, A::Domain>,
}

impl<A: Analysis> Worker<A> {
  fn knowledge(node: &Node<Id>) -> A::Domain

  fn run() {
    let mut worklist = self.flow.nodes;
    while let Some(node) = worklist.pop_first() {
      if self.transfer(node) {
        let next_nodes = self.flow.next_nodes.get(node);
        worklist.extend(next_nodes);
      }
    }
  }

  fn summarize_predecessors(node:&Node<Id>)
    -> A::Domain {
    self.flow
      .predecessors(node)
      .map(|edge| self.analysis
        .transfer(self.knowledge(edge.lhs), edge)
      ).reduce(|a, b| self.analysis.join(a, b))
  }

  fn transfer(node: &Node<Id>) -> bool {
    let old_kw = self.knowledge(node);
    let new_kw = self.summarize_predecessors(node, old_kw);
    self.result.insert(node, new_kw);
    old_kw != new_kw
  }
}
```

### 3.2.1 Reachable nodes

Instead of writing a simple depth-first search algorithm, we can use data flow analysis to calculate the set of all nodes reachable from `begin`, with or without reachability checks.

```
impl Analysis for ReachableNodes {
    type Domain = bool;

    fn bot(&self) -> Domain {
        false
    }

    fn extreme(&self, _program: &Game<Id>) -> Domain {
        true
    }

    fn join(&self, a: Domain, b: Domain) -> Domain {
        a || b
    }
}
```

### 3.2.2 Reaching definitions

This analysis calculates, for every node, which variable assignments can reach that point. At the `begin` node, every variable has its initial value as the only reaching definition. After passing an edge `a, b: x = e`, we update the knowledge about `x`. Note that assignments to maps `a, b: y[x] = e` do not invalidate previous reaching definitions of variable `y`. When joining knowledge coming from multiple edges, the reaching definition of a variable is only known if it was equal at every predecessor node.

### 3.2.3 Reaching assignments

While this analysis is not directly used in optimizations, we need it for calculating `@unique` and `@repeat` pragmas, which have a major influence on the reasoner's speed. *Unique* nodes are those which can only be visited once between each tag. *Repeat* nodes can be visited multiple times without passing through a tag, but only a subset of all variables can change among these visits. By default, every node is *repeat* with all variables. Having the ability to say if a node is *unique* or *repeat* with some variables allows for heavy optimizations in the C++ module and interpreter by avoiding computations in already visited paths. To achieve this, we need to use the *reaching assignments* analysis.

At every node, for every variable, we track which definitions could reach this

place, and which conditions (comparisons and reachability checks) were passed on a path from any of the assignments. A variable assignment is *repeated* if we can reach this assignment multiple times with a non-conflicting set of conditions. That is, conditions leading to this node via different paths are not disjoint. Otherwise, only one of the paths would be legal, so the assignment could be visited only once.

Since this analysis is only used for calculating pragmas, its implementation was not a part of this thesis.

### 3.2.4 Constants analysis

The value of a variable at a given point in the program can sometimes be computed during compilation. For this, we need to keep track of assignments and comparisons, which use constant values. An assignment `a, b: x = e1` updates the knowledge about `x` if the value of `e1` is known at `a`, or cleans it otherwise. Similarly, after a comparison `a, b: x == e2`, if the value of `e2` is constant, we know it will also be the value of `x` at node `b`. Again, when joining knowledge from multiple predecessors, the variable has a constant value only if it was the same constant value at every incoming edge.

This analysis has evolved to also keep track of more complex expressions of known value, e.g., `a, b: y[x] = 1; b, c: y[z] == y[x];`. Using knowledge from the first edge, we can perform an optimization that inlines `y[x]` in the second edge, without removing the assignment.

# Chapter 4

# Optimizations

In this chapter, we list all implemented optimizations, explain their meaning, and compare code listings before and after applying every transformation. We also explain the order in which optimizations run in a fix-point loop, based on the way they enable one another. At the end, we present the removed transformations that were used in a previous language representation.

Transformations implemented in Regular Games can be grouped into five categories:

1. **Expression-oriented:** These optimizations simplify the automaton by propagating constants, inlining variable assignments, merging nested access expressions, and compacting a series of comparisons.
2. **Joins:** These transformations detect local patterns in the automaton, such as forks with common or exclusive actions.
3. **Prunings:** These optimizations remove unused bindings, variables, constants, and unreachable edges.
4. **Reachability-oriented:** The goal of these transformations is to simplify and inline reachability checks when it is legal.
5. **Normalization:** These transformations do not necessarily optimize the game, but make it more standardized. Examples include adding explicit casts in expressions, normalization of constants, and expansion of assignments *any* and variable tags.

Transformations are described in the same order as they run in a fixed-point loop. If any optimization changes the game description, the loop restarts; otherwise, the next transformation is run.

Changing the order of transformations should not change the outcome game in a significant way, but can make the compilation process much longer – if a single

optimization that enables many others is run at the end of the loop, we need to do many additional runs to reach a fixed point. At some point, we reach a place of no return, after which we no longer restart the loop. Transformations that happen after this point only calculate pragmas and do not change the basic game representation itself.

## 4.1 Optimizing transformations

### 4.1.1 Compact skip edges

Most transformations, instead of removing redundant edges, convert them into skip edges. Skip edges are also commonly used by developers to make a cleaner, more readable automaton. Additionally, they appear in large quantities after translation from other languages. This optimization removes them when possible and is run after every other transformation. There are multiple cases when we can remove a skip edge. The first case is called *skip backwards*: if `b, c: ;` is the only outgoing edge from `b`, we can replace all edges `a, b: <1>` with `a, c: <1>`, and remove the skip edge. The second case, called *skip forward*, occurs when `a, b: ;` is the only edge that enters `b`. Then we can replace every edge `b, c: <1>` with `a, c: <1>`, and again remove the skip edge. In both cases, we need to additionally make sure that `b` is not a reachability target. Compact skip edges phase performs one more optimization: if a skip edge `a, b: ;` exists, we can remove all conditional edges between `a, b`, since they are straightforwardly redundant.

```
begin, b: 1 == 1;
b, c: ;                          begin, c: 1 == 1;
c, a1: 2 == 2;                   c, a1: 2 == 2;
```

Figure 4.1: Before and after *compact skip edges*.

### 4.1.2 Inline assignment

This optimization is the opposite of a classical extract common sub-expression transformation. In RG, we care more about the number of assignments than the complexity of expressions. Because of that, it is beneficial to compute the value of the same expression multiple times, if that allows us to remove an unnecessary assignment. To do that, for each edge $\texttt{e}_1$ with an assignment to a variable `x = expr`, we try to replace each reference to `x` with `expr`, until `x` is reassigned. If there are no references to `x`, we can simply remove this assignment. Otherwise, let $A$ be the set of all variables used in `expr`, with the location of their last assignment, if known. To calculate where the last assignment of each variable is, at each point in the program, we need to use data flow analysis *reaching definitions* (3.2.2). For each edge $\texttt{e}_2$ referencing `x`, if the

last definition of `x` was at edge $e_1$, we calculate the current state of variables from $A$ and compare it with the state at edge $e_1$. If the definitions of all variables match at every usage of `x`, we can inline this assignment and skip edge $e_1$ (replace its label with a skip). This optimization runs early in the loop because it enables many other optimizations, namely, it allows us to inline more reachability subautomata.

```
begin, a: coordX = board[x];      begin, a: ;
a, b: coordX != coordY;           a, b: board[x] != coordY;
b, c: coordX = up[coordX];        b, c: coordX = up[board[x]];
```

Figure 4.2: Before and after *inline assignment*.

### 4.1.3 Compact reachability

Subautomata are used only for reachability checks between two nodes, and state changes applied in them are undone when leaving the subautomaton. For these reasons, we can move the ends of reachability checks to omit trailing skip edges, tags, and assignments. Similarly, we can omit leading skip edges and tags. Note that these edges cannot be directly removed because the rechability subautomaton can be embedded in the main automaton. This optimization runs before *inline reachability* to make the inlined automaton as small as possible.

```
a, b: ? x0 → y0;                  a, b: ? x1 → x2;
x0, x1: ;                         x0, x1: ;
x1, x2: coordX == coordY;         x1, x2: coordX == coordY;
x2, y0: ;                         x2, y0: ;
```

Figure 4.3: Before and after *compact reachability*.

### 4.1.4 Inline reachability

This transformation aims to replace a reachability check `? a →b` by the reachability subautomaton between `a` and `b`, i.e., all edges reachable from `a` without going through `b`. Negated reachability checks are inlined if they consist of only one edge (more complex patterns are much harder to negate). If a game contains both checks `? a →b` and `! a →b`, then we inline both or none. This is because the runtime can detect that these checks are disjoint and only perform one of them. This would not be possible if one of the checks were inlined.

Reachability subautomata can contain assignments and tags. The latter can be inlined if the reachability check is not a part of the main automaton (it is not reachable from `begin`). The former poses a bigger problem. After performing a reachability check, the old state before the check is brought back unmodified, and inlining assignments could change that behavior. To make this optimization safe,

we make sure that every variable with an assignment inside the reachability check is reassigned in the original subgraph before being referenced. If all of these conditions occur, we can replace an edge `p, q: ? a →b` with edges `p, a: ;`, `b, q: ;`, and a reachability subatomaton between `a` and `b`.

```
begin, a1: ? a → c;          begin, a: ;
a, b: 1 == 1;                a, b: 1 == 1;
b, c: 2 == 2;                b, a1: 2 == 2;
```

Figure 4.4: Before and after *inline reachability*.

### 4.1.5 Propagate constants

The value of a variable `x` is known at a node `a` when it is either its the initial value of `x` or the single reaching definition of `x` was in form $\texttt{node}_1\texttt{, node}_2\texttt{: x = expr}$, where `expr` value is known at $\texttt{node}_1$. An expression value is known at some node when it consists only of symbols, constants, and variables of value known at this node. If an expression is of a set type and its value is known, we can replace that expression with its value. To check that, we need to use the data flow analysis *constants analysis* (3.2.4). Additionally, if we encounter an assignment `x = expr`, where `x` and `expr` have the same constant value, we can skip this edge. Similarly, if we encounter a comparison $\texttt{e}_1\texttt{ == e}_2$ when $\texttt{e}_1$ and $\texttt{e}_2$ have the same constant value, or $\texttt{e}_1\texttt{ != e}_2$ when they have different constant values, we can skip these comparisons. What is important to note, we can only inline values of a set type, because maps are not a valid expression in RG. This optimization runs after *inline assignment* because they conflict with each other (both change the same parts of the rules), but the latter yields better results.

```
type A = { 1, 2, 3, 4 };
const down: A → A =
  { 4:3, 3:2, :1 };
var x: A = 3;
var y: A = 2;
begin, a: y = A(*);
a, a1:
  y == board[down[x]];
```

```
type A = { 1, 2, 3, 4 };
const down: A → A =
  { 4:3, 3:2, :1 };
var x: A = 3;
var y: A = 2;
begin, a: y = A(*);
a, a1: y == board[2];
```

Figure 4.5: Before and after *propagate constants*.

### 4.1.6 Merge accesses

For each expression with nested accesses $\texttt{map}_1\texttt{[map}_2\texttt{[x]]}$ where $\texttt{map}_1$ is a constant of type `B →C` and $\texttt{map}_2$ is a constant of type `A →C`, we can create a new constant $\texttt{map}_1\texttt{\_map}_2$ of type `A →C` such that $\texttt{map}_1\texttt{[map}_2\texttt{[x]] == map}_1\texttt{\_map2[x]}$ for every `x` in `A`.

The same constant can be reused for every occurrence of nested $\texttt{map}_1\texttt{[map}_2\texttt{[\_]]}$. We can do the same for more complex cases, namely $\texttt{map}_1\texttt{[x][map}_2\texttt{[y]]}$ becomes $\texttt{map}_1\texttt{\_map}_2\texttt{[x][y]}$ and $\texttt{map}_1\texttt{[map}_2\texttt{[x][y]]}$ becomes $\texttt{map}_1\texttt{\_map}_2\texttt{[x][y]}$. This optimization runs after *propagate constants*, because now there is a higher chance of encountering complex expressions with multiple maps of known value.

```
const MapAB: A → B =
  { 1: 1, :2 };
const MapBC: B → C =
  { 1: 2, 2: 3, :4 };
begin, a1:
  x = MapBC[MapAB[1]];
```

```
const MapBC_MapAB:
A → C = { 1: 2, :3 };
begin, a1: x =
  MapBC_MapAB[1];
```

Figure 4.6: Before and after *merge accesses*.

### 4.1.7 Reorder conditions

Considering multiple simple paths starting at a given node $\texttt{node}_1$, if there are no assignments on any of these paths, we can reorder labels on them so that the labels that appear on multiple paths are moved closer to $\texttt{node}_1$. This allows other transformations, specifically *join fork prefixes*, to optimize much more. However, applying this transformation is not always safe. Although reordering conditions will not change the possible moves or outcome states, it can break the game by allowing dangerous accesses that were previously guarded by safety checks. Namely, in Figure 4.7, we can see an expression `z[x]`, but if `var z: T →T` where `type T = { 2,3,4}`, then this map is not defined for the value `1`. This case is detected, and the reordering does not happen.

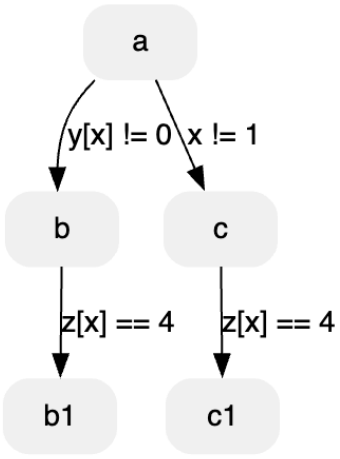


Figure 4.7: Unsafe access guarded by a check on `x`.

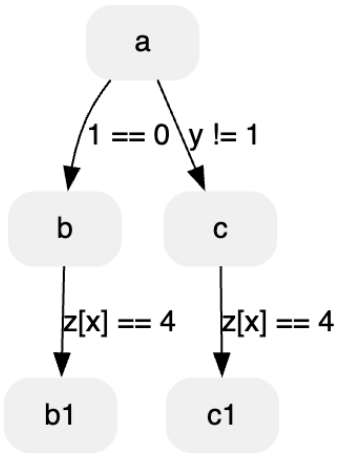


Figure 4.8: Unsafe access guarded by an unrelated expression.

We try to avoid such situations by ensuring that any expression used as an access will not be moved before a check that compares it with some value. Still, this is not enough because of checks that are not directly related to the access expression that appears later, and changing their order can break the game, as in Figure 4.8. In this case, it is left to the game's author to determine if their game uses such checks.

```
begin, a: 4 == 4;
begin, b: 2 == 2;
a, a1: 1 == 1;
b, b1: 4 == 4;
```

```
begin, a: 4 == 4;
begin, b: 4 == 4;
a, a1: 1 == 1;
b, b1: 2 == 2;
```

Figure 4.9: Before and after *reorder conditions*.

If a game description in HRG contains code similar to this in Figure 4.10, this optimization will help to join common conditions. In this example, the same labels appearing in horizontal and vertical checks will be reordered and later merged by *join fork prefixes*.

```
branch I in {0,1,2,3,4,5} {
  branch J in {0,1} {
    branch {
      repeat K in {0,1,2,3,4} {// Vertical
        check(board[P(I, J+K)] == piece)
      }
    } or {
      repeat K in {0,1,2,3,4} {// Horizontal
        check(board[P(J+K, I)] == piece)
      }
    }
  }
}
```

Figure 4.10: Nested loop with alternative in *Pentago.hrg*

### 4.1.8 Compact comparisons

This optimization reduces the automaton size by compressing a set of comparisons into a more efficient form. Given a set of parallel edges between two nodes, we check if multiple among them are comparisons on the same expression `e` in the form of `e == x_i`, where `e` is of set type `T` and `x_i` are symbols of type `T`. Let $S$ be the set of these edges, and $X = \{x_1, x_2, ..., x_n\}$. We resolve the type of expression `e` and calculate the set of its members $A$. Let $Y = T - X$. If $|Y| < |X|$, we can optimize this subautomaton by checking if the value of `e` does not belong to $Y$. To achieve this, we replace the edges `a, b: e == x_1; a, b: e == x_2; ...` with `a, a_1: e != y_1; a_1, a_2: e != y_2; ...a_n-1, b: e != y_n;`.

```
type A =
  { 1, 2, 3, 4, 5};
var x: A = 1;
begin, a: x == 1;
begin, a: x == 2;
begin, a: x == 3;
```

```
type A =
  { 1, 2, 3, 4, 5 };
var x: A = 1;
begin, b: x != 4;
b, a: x != 5;
```

Figure 4.11: Before and after *compact comparisons*.

### 4.1.9 Join exclusive edges

If there exist two edges with comparisons `a, b: e_1 == e_2;` and `a, b: e_1 != e_2;`, or with a reachability check: `a, b: ? p →q;` and `a, b: ! p →q;`, then we know that one of the conditions is always true, so we can replace these edges with a skip edge `a, b: ;`. This transformation also simplifies more general cases, as presented in Figure 4.12. Consider nodes `a` and `b` with multiple conditional edges between them.

Let us call the set of these edges $S$ and the set of their labels as $SL = \{s_1, s_2, s_3, ...\}$. This means that we can get from `a` to `b` if the expression $s_1 \vee s_2 \vee s_3...$ is true. There is also a path $C$ between $a$ and $b$, which consists only of conditional edges and $length(C) \geq 2$. $CL = \{c1, c2, c3, ...\}$ will be the set of edges' labels in $C$.

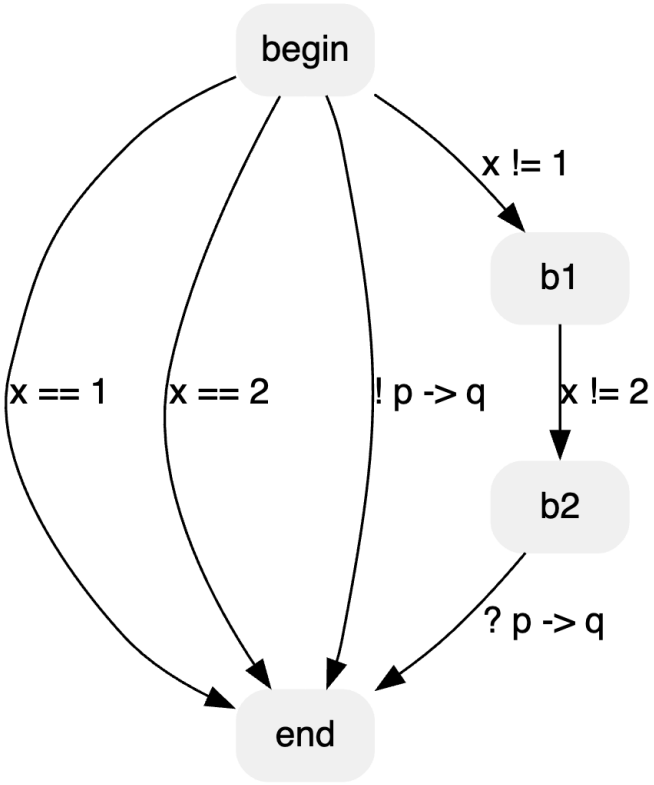


Figure 4.12: Subautomaton with a complex check.

Then we can get from `a` to `b` if the expression $(c1 \wedge c2 \wedge c3...) \vee s1 \vee s2 \vee s3...$ is true. Now, if every condition in $CL$ is negated in $SL$, then this becomes $(c1 \wedge c2 \wedge c3...) \vee \neg c1 \vee \neg c2 \vee \neg c3...$ which is a tautology. So in this case, we can replace all edges in $S$ and $C$ with a single skip edge `a, b: ;`.

```
begin, a: ? p → q;
begin, a: ! p → q;
c, d: x == y;                    begin, a: ;
c, d: x != y;                    c, d: ;
```

Figure 4.13: Before and after *join exclusive edges*.

### 4.1.10 Join fork prefixes

If multiple edges with the same label come out of one node `a`, we can optimize this part of the automaton. Let `a, b: <label>;` be the first of such edges. Now, we

need to ensure that `a, b: <label>;` is the only edge coming into `b`, and that we do not modify any reachability target, otherwise we could change the game's semantics. Then, we can move the rest of these edges to start at `b`, and skip their labels. This way, we might make some paths longer, but we will save unnecessary computations in labels.

```
begin, b: 1 == 1;          begin, b: 1 == 1;
begin, c: 1 == 1;          b, a: 2 == 2;
b, a: 2 == 2;              c, d: ;
c, d: ;                    b, c: ;
```

Figure 4.14: Before and after *join fork prefixes*.

### 4.1.11 Join fork suffixes

Similarly to the previous transformation, we can optimize the case when multiple edges with the same label come into one node `b`. Let `a, b: <label>;` again be the first of such edges. Then, instead of having other edges going to `b` with the same label, we can move their ends to `a` with skip labels. Again, first we need to check if `a, b: <label>;` is the only edge leaving `a`, and that none of the modified paths contains a reachability target.

```
begin, a1: x == 1;
begin, a2: x == 2;          begin, a1: x == 1;
a1, b: 0 == 0;              begin, a1: x == 2;
a2, b: 0 == 0;              a1, b: 0 == 0;
```

Figure 4.15: Before and after *join fork suffixes*.

### 4.1.12 Skip self assignments and skip self comparisons

These are two of the simplest available optimizations. The first replaces self assignments `expr = expr` with skip edges. The latter does the same for self comparisons `expr == expr`.

```
begin, a1: x == x;          begin, a1: ;
begin, a3: x = x;           begin, a3: ;
```

Figure 4.16: Before and after *skip self assignments and comparisons*.

### 4.1.13 Skip unused tags

In reachability subautomata, tags are ignored, so if a tag (or a variable tag) is not reachable from the `begin` node, it can be skipped. This does not apply to artificial

tags, i.e., tags marked with the `@artificialTag t` pragma, which are optimized separately. While this optimization enables *inline reachability*, we run it later in the loop – this way multiple smaller reachability subautomata are merged into one, which later is possibly inlined into the main game loop. If we changed the order, a single reachability check could be inlined into the main automaton, while the others would stay separated.

```
begin, a: ? t1 → t3;          begin, a: ? t1 → t3;
t1, t2: $ 1;                  t1, t2: ;
t2, t3: $ 2;                  t2, t3: ;
```

Figure 4.17: Before and after *skip unused tags*.

### 4.1.14 Skip redundant tags

Skips edges with tags that are not needed when choosing a move - they appear in every possible choice at the same position. If it is a variable tag, the value of this variable must be the same on every path for this tag to be redundant.

```
begin, a: $ t1.               begin, a: $ t1.
begin, b: $ t2;               begin, b: $ t2;
begin, c: $ t3;               begin, c: $ t3;
a, a1: $ rt;                  a, a1: ;
b, a1: $ rt;                  b, a1: ;
c, a1: $ rt;                  c, a1: ;
```

Figure 4.18: Before and after *skip redundant tags*.

### 4.1.15 Prune self loops

This transformation removes loops in the form `x, x: l`, where `label` is anything but an assignment or a tag.

```
a, a: ;
b, b: 1 == 1;
c, c: coord = up[coord];      c, c: coord = up[coord];
```

Figure 4.19: Before and after *prune self loops*.

### 4.1.16 Prune singleton types

Eliminates set types with only one possible value, together with variables of that type.

```
type T = { 1 };
var t: T = 1;
a, b: t = 1;                    a, b: ;
b, c: t == 1;                   b, c: 1 == 1;
```

Figure 4.20: Before and after *prune singleton types*.

### 4.1.17 Prune unreachable nodes

This removes edges and nodes that are not reachable from the `begin` node or the start of a reachability check. Also filters out dead ends, that is, nodes with no outgoing edges, that are not the `end` node or a reachability target.

```
begin, b: ? r1 → target;
b, end: ;
a, end: ;                       begin, b: ? r1 → target;
r1, r2: ;                       b, end: ;
r1, target: ;                   r1, target: ;
```

Figure 4.21: Before and after *prune unreachable nodes*.

### 4.1.18 Prune unused constants and variables

This removes constants and variables that are not used in any expression, as a tag variable, nor in definitions of other constants and variables.

### 4.1.19 Skip artificial tags

This skips edges with tags marked with the `@artificialTag t` pragma. Artificial tags exist to ensure we do not optimize some parts of the game description, which helps to detect *unique* nodes. This optimization runs at the very end, so that no further transformations take place after artificial tags are removed.

```
begin, a1: $ t;                 begin, a1: ;
@aritificialTag t;              @aritificialTag t;
```

Figure 4.22: Before and after *skip artificial tags*.

## 4.2 Normalization

### 4.2.1 Normalize constants

This makes map definitions at most one level deep. Also, moves variable default values to constants.

```
var X:
Bool → Bool → Bool
  = { :{ :0 } };
```

```
const Hoisted_1:
Bool → Bool
  = { :0 };
const Hoisted_2:
Bool → Bool → Bool
  = { :Hoisted_1 };
var X:
Bool → Bool → Bool
  = Hoisted_2;
```

Figure 4.23: Before and after *normalize constants*.

### 4.2.2 Normalize types

This makes set types appear only as a type definition. Also, it makes them at one most level deep.

```
var X:
Bool → Bool → Bool
  = { :{ :0 } };
```

```
type Type1 = Bool → Bool;
type Type2 = Bool → Type1;
var X: Type2 = { :{ :0 } };
```

Figure 4.24: Before and after *normalize types*.

### 4.2.3 Add explicit casts

Infers the type of every subexpression and adds a type cast to it, even if the conversion is trivial.

```
type T = { 1 };
var t: T = 1;
a, b: t == t;
```

```
type T = { 1 };
var t: T = 1;
a, b: T(t) == T(t);
```

Figure 4.25: Before and after *add explicit casts*.

### 4.2.4 Expand assignment any

For each edge with assignment `x = A(*)` and for each member of type `A`, it creates a new edge assigning that member to `x`.

```
type Coord = { 0, 1, 2 };
a, b: coordX = Coord(*);
b, c: coordY = Coord(*);
```

```
type Coord = { 0, 1, 2};
a, b: coordX = 0;
a, b: coordX = 1;
a, b: coordX = 2;
b, c: coordY = 0;
b, c: coordY = 1;
b, c: coordY = 2;
```

Figure 4.26: Before and after *expand assignment any*.

### 4.2.5 Expand tag variable

For each edge with tag variable `$$ x`, where `x` is a variable of type `A`, and for each member `a` of type `A`, it creates a new edge with tag `$ a`, preceded with comparison `x == A(a)`.

```
type Coord = { 0, 1, 2 };
a, b: $$ coord;
```

```
type Coord = { 0, 1, 2 };
a, b0: coord == Coord(0);
b0, b: $ 0;
a, b1: coord == Coord(1);
b1, b: $ 1;
a, b2: coord == Coord(2);
b2, b: $ 2;
```

Figure 4.27: Before and after *expand tag variable*.

### 4.2.6 Mangle symbols

Replaces symbol names with new, unique, and short values. It does not change the names of built-in symbols.

```
begin, a1:
  playerTurn = Player(X);
turn, move: ? a → b;
```

```
begin, a1:
  _t = Player(X);
_w, _u: ? _u → _v;
```

Figure 4.28: Before and after *mangle symbols*.

## 4.3 Obsolete optimizations

Constructs *any assignment* and *variable tag* have replaced an older representation that used *generator nodes* and *bindings*. A generator node, denoted as `a(t: T)`, uses a binding `t` of type `T`. A path where every node has the same binding expands to a set of paths, where on each of them the bound variable is replaced by a different value of type `T`. While this representation could be considered more readable, it made the analysis and optimization stages of the compilation process much more complicated. For this reason, it was replaced by the aforementioned syntax.

Before that, there were multiple transformations related to bindings. Furthermore, most of the other optimizations had additional checks to prevent accidentally merging unrelated bindings or breaking generator nodes. In Figure 4.29, we cannot perform the *compact skip edges* optimization, because it would connect two different bindings with the same name, thus changing the semantics.

```
type Coord = { 0, 1, 2 };
a(c: Coord), b: c == 1;
b, d(c: Coord): ;
d(c: Coord), a1(c: Coord): c == 2;
```

Figure 4.29: Two unrelated bindings with the same name.

### 4.3.1 Expand generator nodes

For each node with a generator of type `A` and for each member of type `A`, creates a new node replacing the generator with that type member.

```
type Coord = { 0, 1, 2 };
a, a(cX:Coord):
  $ cX;
a(cX:Coord), c:
  posX = Coord(cX);
```

```
type Coord = { 0, 1, 2};
a, a0: $ 0;
a, a1: $ 1;
a, a2: $ 2;
a1, c:
  posX = Coord(0);
a2, c:
  posX = Coord(1);
a3, c:
  posX = Coord(2);
```

Figure 4.30: Before and after *expand generator nodes*.

### 4.3.2 Join generators

If multiple paths consisting of a comparison with the same variable `x` followed by an assignment `a(bind`$_1$`: Type`$_1$`), b: x = bind`$_1$`;` come out of one node, this transforma-

tion replaces them with a single path. For that, it generates a fresh constant that checks which paths were reachable for a given value of `x`.

```
type All = {                    type All = {
  1, 2, 3, 4, 5, 6 };             1, 2, 3, 4, 5, 6 };
type Type1 = { 3, 4, 5 };       type Type1 = { 3, 4, 5 };
type Type2 = { 1, 2, 3 };       type Type2 = { 1, 2, 3 };
type Type3 = { 1, 3, 6 };       type Type3 = { 1, 3, 6 };
var coord : All = 1;            type J1 = {
begin, a1(v1: Type1):             1, 2, 3, 4, 5, 6 };
  coord == All(1);              const j1:
begin, a2(v2: Type2):           All → J1 → Bool = {
  coord == All(2);                1:{3:1, 4:1, 5:1, :0},
begin, a3(v3: Type3):             2:{1:1, 2:1, 3:1, :0},
  coord == All(3);                3:{1:1, 3:1, 6:1, :0},
a1(v1: Type1), b:                 :{ :0 } };
  coord = v1;                   var coord: All = 1;
a2(v2: Type2), b:               begin, a1(b_J1: Joined1):
  coord = v2;                     j1[coord][b_J1] == 1;
a3(v3: Type3), b:               a1(b_J1: Joined1), b:
  coord = v3;                     coord = bind_J1;
```

Figure 4.31: Before and after *join generators*.

### 4.3.3 Merge bindings

It is a common pattern that we assign a binding value $(\texttt{bind}_1\texttt{: T})$ to a variable $\texttt{x = bind}_1$ and later, when introducing another binding $(\texttt{bind}_2\texttt{: T})$, compare its value with the variable $\texttt{bind}_2\texttt{ == x}$. Using *reaching binding assignments* analysis, we can detect at any point in the program if the last assignment to each variable was to a binding. Then, when introducing a new binding in an edge $\texttt{a, b(bind}_2\texttt{: T}_2\texttt{): bind2 == x;}$, if $\texttt{T}_1\texttt{ = T}_2$, last assignment to `x` was $\texttt{x = bind}_1$, and there were no binding introduced between $\texttt{bind}_1$ and $\texttt{bind}_2$, we can merge them into one, by propagating $\texttt{bind}_1$ through all edges until the definition of $\texttt{bind}_2$.

```
begin, a1(v1: Coord):
  1x == board[x];               begin, a1(v1: Coord):
a1(v1: Coord), a2:                v1 == board[x];
  coord = v1;                   a1(v1: Coord), a2(v1: Coord):
a2, a3(x2: Coord):                coord = v1;
  x2 == coord;                  a2(v1: Coord), a3(v1: Coord);
```

Figure 4.32: Before and after *merge bindings*.

### 4.3.4 Skip generator comparisons

Another common pattern in translated games was introducing a generator `(bind: T)`, comparing it with some value of type `T`, `bind == expr`, and never referencing this generator again. These bindings can be removed because there is always a single value for which this comparison is true.

```
a, b(t: T): t == null;
b(t: T), c: ;
```

```
a, b(t: T): ;
b(t: T), c: ;
```

Figure 4.33: Before and after *skip generator comparisons*.

# Chapter 5

# Results

Table 5.1 lists how many game descriptions were used for calculating the results. Most games were translated from RBG, and these include mostly games under 500 edges and between 2,000 and 4,000 edges. The biggest game in the dataset is *Pentago.rbg*, with over 10,000 edges. The smallest games come from HRG, with 12 games under 100 edges. A lot of games in HRG with a size between 100 and 500 edges were written using LineGames. We have also used three games translated from GDL.

Table 5.1: Number of games of each size, in the number of edges before optimizations.

| | **Number of edges (unoptimized)** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | <50 | <100 | <500 | <1,000 | <2,000 | <4,000 | >4,000 | Total |
| RBG | - | - | 29 | 4 | 12 | 23 | 7 | 75 |
| GDL | - | - | 1 | 1 | 1 | - | - | 3 |
| HRG | 4 | 8 | 42 | 4 | 2 | 2 | 1 | 63 |
| Total | 4 | 8 | 72 | 9 | 15 | 25 | 8 | 141 |

## 5.1 Dependencies between transformations

As explained in the previous chapters, optimizations depend on each other. One optimization can enable another, but might also conflict with a different one. Figure 5.1 shows how turning on one transformation affects other optimizations.

First, we notice that some key transformations have a big influence on all other optimization phases. These include *compact skip edges*, *inline assignment*, *inline reachability*, and *join fork prefixes*. Some other transformations work in pairs. *Reorder conditions* enables *join fork prefixes* by moving edges with the same labels, *propagate constants* provides work for *skip self comparisons*, *prune unused variables* and *constants* naturally cooperate.

The most important transformation, *compact skip edges*, affects almost every

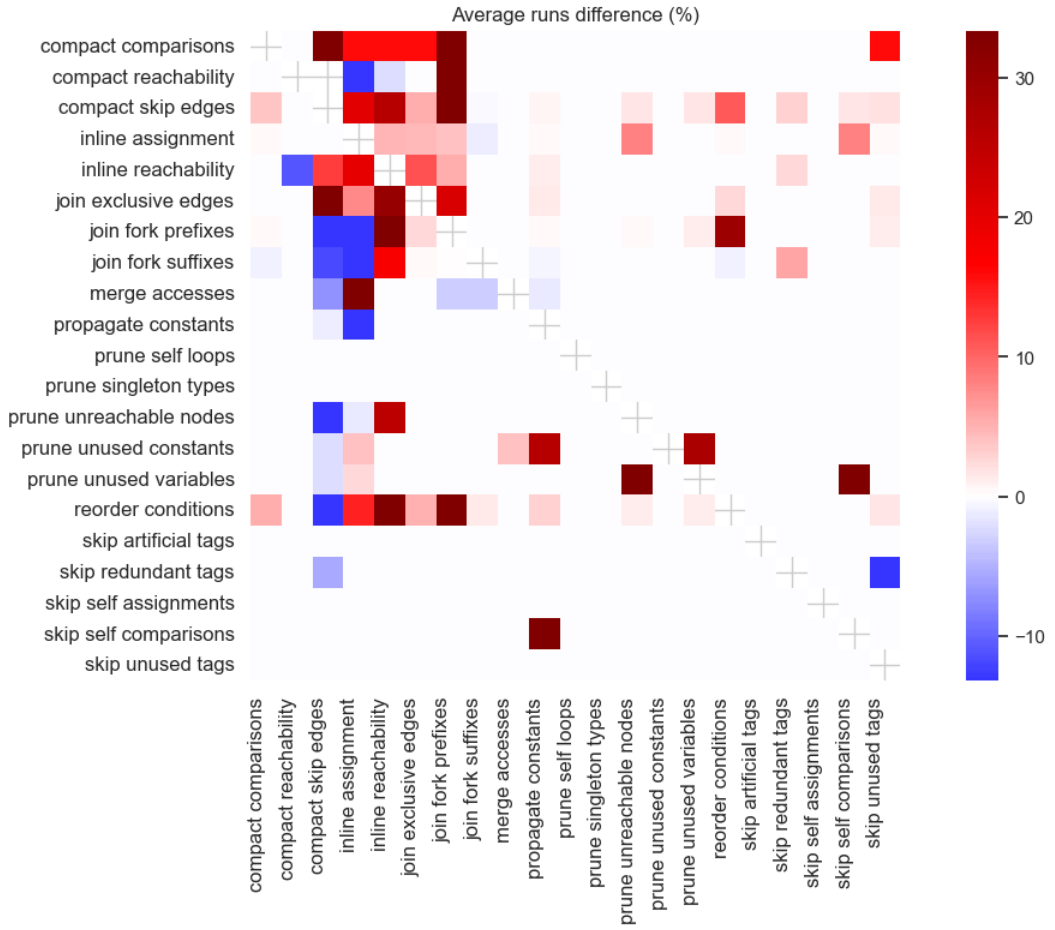


Figure 5.1: How optimizations enable one another. Each cell indicates how many times more often (in %), on average, the transformation on the y-axis was applied when the corresponding transformation on the x-axis was turned on, compared to when it was off.

other one, but in a different way than the other key optimizations. While they usually allow other phases to run more often, *compact skip edges* makes other transformations run fewer times. This is because it significantly reduces the size of the automaton, and the number of times each optimization runs is related to the number of nodes. However, this does not mean that they optimize less, only that they can now change more in a single pass. At the same time, *compact comparisons*, *join exclusive edges*, and *inline reachability* run more often. The first two of these look for parts of an automaton where all edges are either comparisons or reachability checks, and thus removing skip edges makes such subgraphs more common.

Next, consider *inline assignment*. We can see in the Figure 5.1 that it enables *inline reachability* and *merge accesses*. The former needs to ensure that no state change inside a reachability subautomaton escapes into the main part, and inlining assignments makes this happen more often. The latter requires complex expressions to run, and they typically do not occur naturally in games. Most of them are a

result of inlining variables. On the other hand, *inline assignment* conflicts with *compact reachability*, and *propagate constants*. The second one needs assignments with constant values to run, and they no longer appear after inlining the assignments.

The third most influential optimization is *inline reachability*. Most reachability subautomata form a straight path with multiple comparisons, and some of them appear on multiple paths. When inlining reachability checks, more complex subgraphs appear and can later be optimized with *reorder conditions*, *join fork prefixes*, and *join exclusive edges*.

## 5.2 Optimizations' influence on metrics

We can measure how an optimization influences the game description in various ways. We decided to select a set of metrics that describe how each transformation makes game descriptions more efficient. They include: the number of edges of each type, the number of nodes, the average branching factor, the number of variables and constants, the number of reachability subautomata, and a statistic very important for performance, state size (the amount of memory required to store the game's state). Figure 5.2 presents the effect of every transformation on these metrics. To assess the contribution of individual optimizations, we conducted ablation studies in which each transformation was disabled in isolation while keeping all others active. Again, we can notice a few key phases that have a great impact on numerous metrics.

The first of them is *compact skip edges*, which highly reduces the number of edges and nodes, at the same time increasing the branching factor. Transformations that minimize the game state size are *inline assignments* and *prune unused variables*. The former is one of the only optimizations that reduce the number of assignment edges, which also allows for decreasing the number of reachability subautomata. This happens because reachability checks are easier to inline if there are no assignments in them. Still, the biggest impact on this metric has *inline reachability*. *Join fork prefixes* also reduces the number of nodes and edges, but not as much as *compact skip edges*. While *merge accesses* increases the number of constants by creating new, complex maps, it reduces the number of *access* expressions, yet this metric is not measured here. *Prune unreachable nodes* is influential because it removes leftover edges from inlined reachability checks. The number of tag edges is naturally impacted by *skip artificial*, *redundant*, and *unused tags*. The first of these optimizations also increases the number of skip edges, because it is run after the point of no restart in the fixed-point loop.

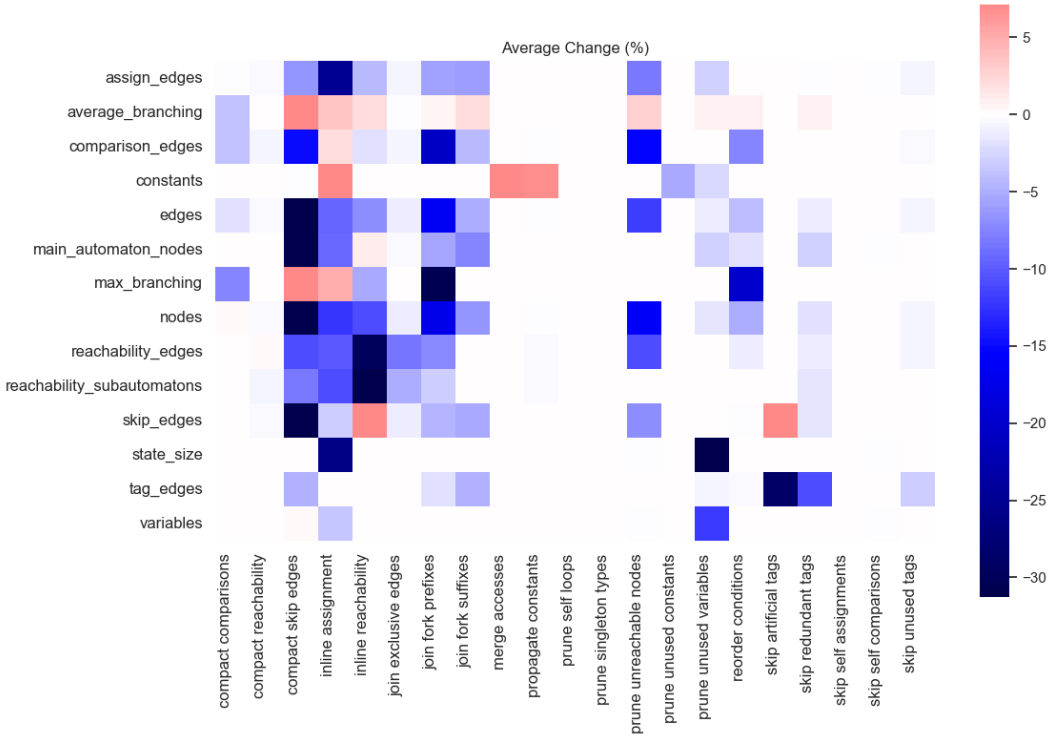


Figure 5.2: The impact of optimizations on metrics. Each cell indicates how the optimization on the x-axis influences (in %), on average, the metric on the y-axis.

## 5.3 Time spent on optimizations

One of the biggest limitations of the Regular Games pipeline is long optimization times on very big games. Table 5.2 shows how long, in milliseconds, the optimization process takes on average for games of different sizes.

Table 5.2: Average optimization time for games (in milliseconds), grouped by the number of edges.

| | **Number of edges (unoptimized)** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | <50 | <100 | <500 | <1,000 | <2,000 | <4,000 | >4,000 | Average |
| RBG | - | - | 72.5 | 271.6 | 759.8 | 10054.0 | 5974.0 | 3804.9 |
| GDL | - | - | 76.8 | 250.1 | 1855.5 | - | - | 727.5 |
| HRG | 4.9 | 13.4 | 68.6 | 189.2 | 711.9 | 892.6 | 3966.1 | 173.7 |
| Avg | 4.9 | 13.4 | 70.3 | 232.5 | 826.4 | 9321.1 | 5723.1 | 2117.0 |

For games under 500 edges (before optimizations), the feedback is almost instant. But for longer descriptions, mostly visible in GDL, the times are significantly prolonged. For games over 4,000 edges, time is already counted in seconds. More complex games with over 10 thousand rules often do not pass the compilation phase in under 5 minutes. Because of this, the future work around optimizations in RG will focus on reducing compilation times, instead of implementing new transformations.

Fortunately, most game descriptions are rather short, and for them, the optimization stage takes under one second.

Figure 5.3 shows what percentage of time spent on each transformation is utilized well – is spent on passes that make changes to the game description. As we can see, some transformations give good results in most languages, namely *compact skip edges*, *join fork prefixes and suffixes*, and *skip artificial tags* (although this optimization does not run for games translated from GDL, because they do not contain artificial tags). While some transformations spend almost no time on effective passes, this is because they run rarely when other, more complex transformations are enabled. Their impact on compilation time is negligible.

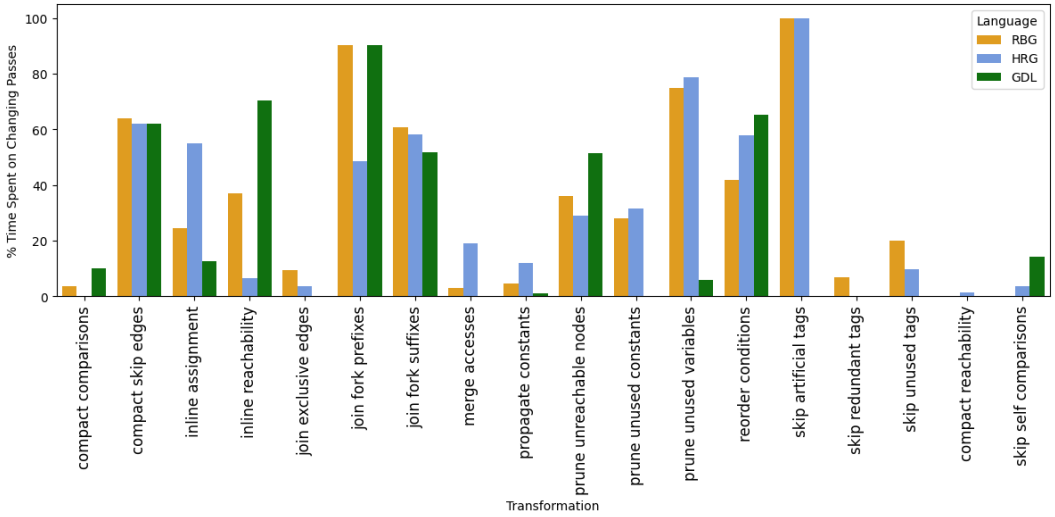


Figure 5.3: Percentage of time spent on each transformation is spent on effective passes.

Figure 5.4 presents what percentage of the whole optimization phase time is spent on each transformation. With this data, we can identify bottlenecks and transformations worth optimizing in our future work. Mainly, *inline assignments* phase takes almost 60% of time in GDL, with a little part of that being well spent. The results look similar for RBG, with over 30% of time spent on this phase. *Skip redundant tags* is effective in RBG, but in HRG it could be entirely disabled. *Propagate constants* is most useful when *inline assignments* phase is turned off; otherwise, we could also disable it to shorten compilation times by 10% in all languages.

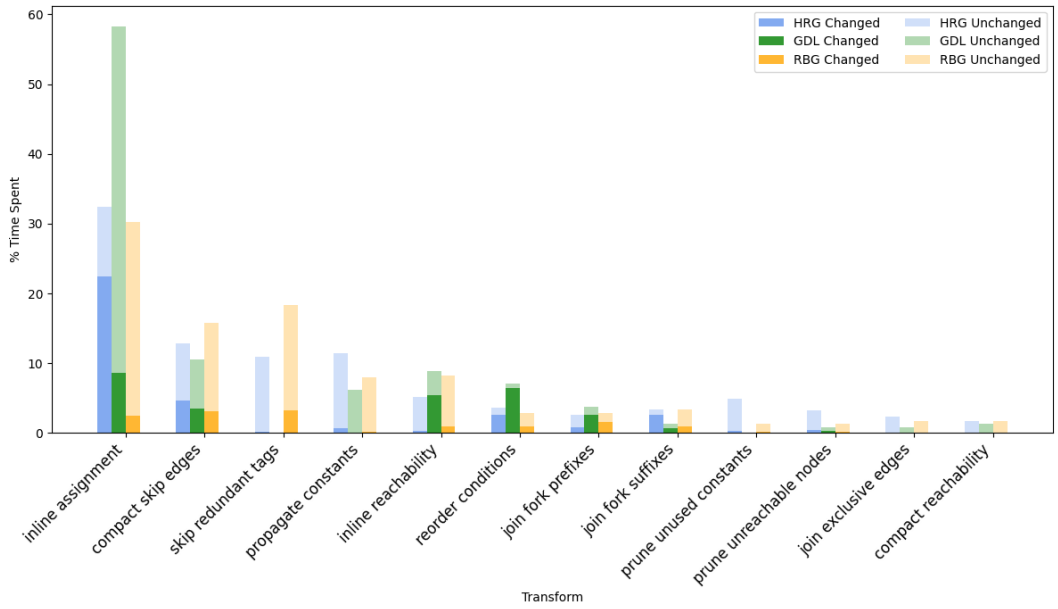


Figure 5.4: Percentage of time spent on each transformation is spent on effective passes.

## 5.4 Impact on the reasoner's performance

Table 5.3: Increase in the number of plays (in %) per 10s, with all optimizations. Measured using the RG interpreter CLI. Environment: MacOS 15.6, Apple M2, 24GB, Rust 1.86.0.

| | **Number of edges (unoptimized)** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | <50 | <100 | <500 | <1,000 | <2,000 | <4,000 | >4,000 | Avg |
| RBG | - | - | 1207.2 | 553.0 | 68.1 | 245.4 | 725.9 | 668.4 |
| GDL | - | - | 59.5 | 50.8 | 45.0 | - | - | 51.8 |
| HRG | 4.9 | 19.1 | 41.7 | 28.1 | 48.4 | 42.2 | 81.1 | 39.7 |
| Avg | 4.9 | 19.1 | 501.6 | 263.9 | 62.4 | 229.1 | 645.3 | 363.4 |

Table 5.3 presents the influence of optimizations on the number of game plays made by the reasoner in 10 seconds. As we can see, the increase varies between source languages and description sizes.

For RBG, the results do not seem to follow any pattern. Instead, we need to look into which games are in each category. The smallest games are mostly variations of *Hex*, *Gomoku*, and *Breakthrough*. The first two respond to optimizations especially well and are the cause of such a great increase. The efficiency of *Breakthrough* games rose by around 400%. In the categories of games under 1,000 edges and 4,000 edges, we have *Amazons* and *Alquerque*, which also strongly benefit from the transformations. Some of the biggest games are variations of chess. Here, the impact

is not that big. This can be the effect of these descriptions being already minimized in Regular Boardgames. The biggest game in the dataset, *Pentago*, is 25 times faster with optimizations.

For High-level Regular Games, we can see that the impact is getting bigger as the game grows. In smaller games, there is not much to optimize, so the difference in performance is small. The sample of GDL games is too small to provide precise results, but similarly to RBG, the influence of optimizations seems to increase for bigger games. But again, the number of plays in 10s for games translated from GDL is too small for these results to be highly reliable.

In conclusion, various game types respond differently to transformations. The impact is always visible for longer descriptions, but even shorter games contain patterns that can be optimized to greatly improve the performance.

# Chapter 6

# Summary and Further Development

## 6.1 Conclusion

This work presents an optimization stage in the Regular Games compiler. First, we have introduced the General Game Playing field, together with the Regular Games language. We have described the compilation process implementation, including the translation from other languages. We have explained how the *data-flow analysis* framework is used in the compiler, and which analyses were implemented.

Next, we have presented the optimization stage by listing all available optimizations. We have described each transformation and justified its position in the fixed-point loop by showing its interaction with other phases. We have shown how every optimization affects the game description.

Finally, we have demonstrated collected results about the impact of transformations. These include the total effect of optimizations on the reasoner's efficiency, altogether, and for games translated from HRG, RBG, and GDL languages. The results also show how each phase influences different metrics about the game's description. We can also clearly see how transformations cooperate and enable each other. Thanks to this data, we can continue our work to improve the efficiency and shorten compilation times.

## 6.2 Future work

This work can be extended in multiple ways. We could implement new, even more complex optimization phases. The problem here is that they would make the compilation process even longer, without improving the efficiency of the game's description by a lot. Most reducible patterns in games are already detectable, and we can only

generalize some transformations to cover more complex cases.

A more reasonable approach would be to focus on transformation performance to reduce compilation time. There are multiple suitable approaches. First, we can optimize each optimization one by one, starting from those that take the longest (*inline assignments* and *skip redundant tags*). Next, the fixed-point loop can be significantly improved. Instead of restarting the loop after each change and rerunning all optimizations, we can rerun only the freshly enabled transformations. For example, there is no point rerunning *inline reachability* after *prune unused constants*. We already know the interactions between phases and can use this to reduce compilation time drastically. What is more, we can implement a solution known from GCC [1], and group optimizations into bundles ( $-O$, $-O_1$, $-O_2$, etc.), where the first one would run only the most time-efficient transformations, and the set of optimizations enabled would grow with each level. This could be a great use for developers who seek a compromise between the game's description efficiency and short compile time.

Alternatively, we can shift our attention away from RG and focus on HRG instead. Implementing optimization phases directly in HRG will benefit from the minimal size of the game's description, compared to the size after translation to RG. Some patterns are also much easier to spot here, because of a syntax more familiar to programmers. Because optimizations in RG have been developed to suit games translated from various languages, not all of them benefit games coming from HRG. Here, we do not have to worry about that. Lastly, since HRG resembles a general-purpose programming language, we can more directly reimplement optimizations used in other compilers.

## Acknowledgments

This work was supported by the National Science Centre, Poland, under project number 2021/41/B/ST6/03691.

[1] `https://gcc.gnu.org/`

# Bibliography


[1] M. Genesereth and M. Thielscher. *General Game Playing*. Morgan & Claypool, 2014.

[2] Michael Genesereth and Nathaniel Love. General Game Playing: Game Description Language Specification. *Technical report, Stanford Logic Group*, 2006.

[3] Michael Thielscher. A general game description language for incomplete information games. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, page 994–999. AAAI Press, 2010.

[4] Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI Mag.*, 26(2):62–72, June 2005.

[5] Éric Piette, Dennis J. N. J. Soemers, Matthew Stephenson, Chiara F. Sironi, Mark H. M. Winands, and Cameron Browne. Ludii - The Ludemic General Game System. *CoRR*, 2019.

[6] Jakub Kowalski, Jakub Sutowicz, and Marek Szykuła. Regular Boardgames. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2017.

[7] H. Gruber and M. Holzer. Finite Automata, Digraph Connectivity, and Regular Expression Size. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP*, pages 39–50. Springer, 2008.

[8] Jakub Cieśluk. Ide for regular games. Eng. thesis, University of Wrocław, 2024.

[9] J. Nievergelt. On the automatic simplification of computer programs. *Commun. ACM*, 8(6):366–370, June 1965.

[10] Edward S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, January 1969.

[11] W. M. McKeeman. Peephole optimization. *Commun. ACM*, 8(7):443–444, July 1965.

[12] A. V. Aho and J. D. Ullman. Transformations on straight line programs - (preliminary version). In *Proceedings of the Second Annual ACM Symposium*

*on Theory of Computing*, STOC '70, page 136–148, New York, NY, USA, 1970. Association for Computing Machinery.

[13] Paul B. Schneck. A survey of compiler optimization techniques. In *Proceedings of the ACM Annual Conference*, ACM '73, page 106–113, New York, NY, USA, 1973. Association for Computing Machinery.

[14] Thomas J. Watson IBM Research Center, F.E. Allen, and J. Cocke. *A Catalogue of Optimizing Transformations*. IBM Thomas J. Watson Research Center, 1971.

[15] Alfred V. Aho, Ravi Sethi, and J. D. Ullman. A formal approach to code optimization. In *Proceedings of a Symposium on Compiler Optimization*, page 86–100, New York, NY, USA, 1970. Association for Computing Machinery.

[16] Radosław Miernik, Marek Szykuła, Jakub Kowalski, Jakub Cieśluk, Łukasz Galas, and Wojciech Pawlik. Regular games – an automata-based general game playing language. 2025.

[17] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, page 194–206, New York, NY, USA, 1973. Association for Computing Machinery.

[18] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. *SIGPLAN Not.*, 37(1):270–282, January 2002.

[19] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, page 1–19, New York, NY, USA, 1970. Association for Computing Machinery.