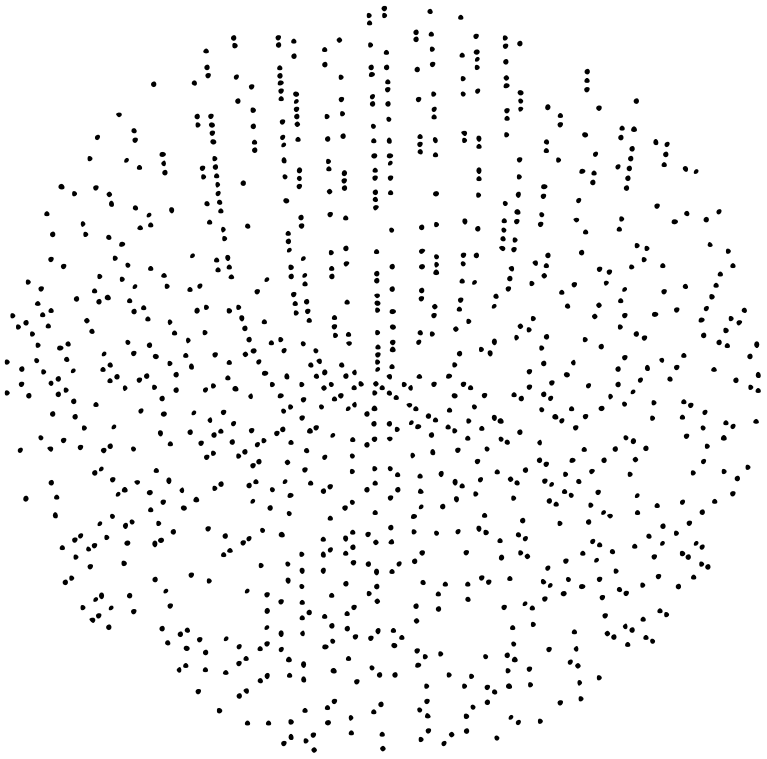
The Static Semantics of the GREEN-MARL Graph Analysis Language

Formal Specification, Declarative Implementation and Integration with a Compiler Back-end



Jeff Smits

The Static Semantics of the GREEN-MARL Graph Analysis Language

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jeff Smits
born in Moordrecht, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: The first 1000 prime numbers mapped on an involute spiral.

The Static Semantics of the GREEN-MARL Graph Analysis Language

Author: Jeff Smits
Student id: 4104889
Email: jeff.smits@gmail.com

Abstract

GREEN-MARL is a domain-specific language for efficient graph analysis. In this thesis, we define the formal static semantics of the language and provide an implementation in the Spoofax language workbench. The type system of GREEN-MARL includes limited forms of name-dependent types, overloading, parametric polymorphism, and inference. We give a formal specification that covers all aspects of this type system. We also describe our implementation of the type system in the Spoofax language workbench, where we focus on the capabilities of Spoofax's meta-languages to describe the type system. GREEN-MARL provides several parallel language constructs, as well as constructs to mitigate data races that can occur in parallel regions. We give a formal description of a symbolic, tree-based dependence analysis that can check the invariants of the mitigation strategies and find potential data races. We employ a rewrite system for the implementation of this analysis in the Spoofax language workbench. Finally, we discuss the integration of these analyses with successive program transformation steps. Each transformation step is informed by the static analysis. However, transformation steps invalidate parts of the analysis results, which inhibits the successive steps. A naive approach to reanalyse the program after every transformation step does not scale. Therefore, we incrementally update analysis results after each transformation step.

Thesis Committee:

Chair:	Prof. Dr. E. Visser	Delft University of Technology
Supervisor:	Dr. G.H. Wachsmuth	Delft University of Technology
Committee member:	Prof. Dr. J.J. Vinju	Eindhoven University of Technology

Preface

Dear reader,

If you're looking for an overview of the thesis, please refer to the outline at the end of Chapter 1. This preface solely contains the acknowledgements.

I would like to thank Guido Wachsmuth so very much for his advise and guidance throughout this thesis project. Guido, your enthusiasm for the subject, your tireless effort to improve my writing, your calm acceptance of sudden changes to the planning, it has all been invaluable to me.

Eelco Visser, thank you for recommending that book on academic writing, which probably prevented me from driving Guido mad. And thanks for suggesting the original subject that lead to this thesis, and for proposing the internship with Oracle Labs, and for vouching for me.

Another thank you to Gabriël Konat and Casper Bach Poulsen for proofreading chapters and providing last-minute feedback.

Hassan Chafi, thank you for taking me on at Oracle Labs. For believing in my ability. And for making no big deal out of my disbelief of the, retrospect completely accurate, prediction of the time required for the internship assignment.

I would also like to thank Sungpack Hong, for his guidance during my internship. And Martin Sevenich for working with me. And really, the entire PGX team for being such warm and welcoming people.

Finally I would like to thank my family for their support. For the way they raised me, where grades in school did not matter, as long as I tried.

Jeff Smits

Moordrecht, the Netherlands

2016-02-02

Contents

List of Figures

Chapter 1

Introduction

Domain-specific languages are programming languages that are tailored to a specific domain. Through the use of terms from this domain, a domain expert can describe her problem and solution succinctly, without spending effort on an encoding in general purpose programming constructs. Programs written in a domain-specific language can be checked for domain-specific properties [37].

This thesis is about GREEN-MARL, a domain-specific language for efficient graph analysis [23]. The language supplies domain specific, high-level features to allow the user to write graph algorithms in a natural and concise way. Language constructs to explicitly specify data-level parallelism allow the user to expose parallelism in these algorithms. GREEN-MARL programs can be compiled to different execution platforms.

Currently, the GREEN-MARL compiler is manually implemented in C++ by a one-man team. This development is becoming harder as the language and the compiler grow. This is not an uncommon situation for a domain-specific language. Domain-specific languages usually have a smaller user base, simply by virtue of being domain specific. Because of this smaller user base, there are typically limited resources available for the development and implementation of the language. The combination of manual implementation, non-trivial features, and small team size hamper development and the exploration that can help to evolve the language.

Language workbenches are suites of tools for programming language development, that promise exploratory language design. The key idea of language workbenches is that they allow a faster definition of a programming language, from which they generate tools such as editors, compilers and interpreters [18]. This is done through more high-level language declarations, that decrease language development and maintenance effort and increase language extensibility. Common examples of languages workbenches are XText [15], MPS [27] and Spoofax [43]. In this thesis we present a re-implementation of the GREEN-MARL compiler in Spoofax.

In order to implement GREEN-MARL properly, we do need to understand its semantics, which is currently given by informal text and implicitly by the implementation. In this thesis, we focus on the formal static semantics of GREEN-MARL, because the language has a number of non-trivial features. The type system has limited forms of name-dependent types, overloading, parametric polymorphism, and inference. The language has several parallel language constructs, as well as

constructs to mitigate data races that can occur in parallel regions. A dependence analysis is needed to check the invariants of the mitigation strategies and find potential data races. We present a formal specification of both the type system and the dependence analysis.

1.1 Research Questions

The challenges of understanding the static semantics of GREEN-MARL and implementing it in a language workbench leads us to the following research questions.

RQ1. What is the static semantics of GREEN-MARL?

The GREEN-MARL language specification [22] is written in prose. The specification of the language is in some places unclear, incomplete, and inconsistent. To fully understand the language, we need to pin down its static semantics formally. This is particularly important because the domain specific nature of GREEN-MARL's type system. A formal specification can be used to discuss GREEN-MARL's semantics with confidence and provides the foundation for implementations of the language.

RQ2. How can the static semantics be declared in Spoofax?

Formal type systems are typically not executable. Based on this type system, we need to implement language processors such as a type checker, type analysis for further compilation steps, and reference resolution for an editor. The Spoofax language workbench can generate all these features from a specification of the type system in its declarative meta-languages NABL [29] and TS. The type system includes limited forms of name-dependent types, overloading, parametric polymorphism and inference, which may not fit the constraints of the meta-languages. We explore if the capabilities of the meta-languages are sufficient to describe GREEN-MARL's type system and search for workarounds when they are not.

RQ3. What is the formal semantics of the dependence analysis of GREEN-MARL?

GREEN-MARL provides several parallel language constructs, as well as constructs to mitigate data races that can occur in parallel regions. To validate the proper use of those mitigation strategies, and detect data races, the language needs a dependence analysis. Currently, this analysis is only described by example [23] and in a C++ implementation. For a proper understanding of this analysis, we need a formal specification. Based on this formal specification, implementations can be built.

RQ4. How can this dependence analysis be declared in Spoofax?

Again, the formal specification of such an analysis is typically not executable. Rewrite systems are a common way to express static analyses [4]. Spoofax provides the trans-

formation language STRATEGO[1]. We explore how we can fit the dependence analysis of GREEN-MARL into STRATEGO.

RQ5. How can analysis results be kept consistent after transformations?

The GREEN-MARL compiler performs successive program transformation steps for optimisations. Each transformation step is informed by the static analysis. However, each transformation step invalidates parts of the analysis results, which inhibits the next step. A naive approach to re-analyse the program after every transformation step does not scale. Therefore we seek an approach to incrementally update analysis results after each transformation step.

1.2 Contributions

The contributions of this thesis are:

- A formal specification of GREEN-MARL's type system (Chapter 3).
- A formal specification of a dependence analysis for GREEN-MARL (Chapter 4).
- The implementation of this static semantics in Spoofax (Chapter 5 and Sections 5.2,5.3).
- An approach to update analysis results specific to this implementation (Chapter 5, Section 5.4), to integrate this implementation in a full Spoofax-based GREEN-MARL compiler.

1.3 Outline

The remainder of the thesis is structured as follows: we introduce GREEN-MARL in detail in Chapter 2 with two real world algorithm implementations. Through these examples, we illustrate most of the language features, the type system and the dependence analysis. We follow this up by our first contribution: a formal static semantics of GREEN-MARL (Chapter 3). We use formal type rules to capture the unique qualities of GREEN-MARL's domain specific features. In Chapter 4 we describe the dependence analysis. We describe our implementation of GREEN-MARL's name and type semantics in NABL and TS, the implementation of the dependence analysis in STRATEGO, and the challenges we found along the way (Chapter 5). We end with related work (Chapter 6) and discussion (Chapter 7).

[1] The citation is on Term Graph Rewrite Systems, but the same applies for Term Rewrite Systems like STRATEGO.

Chapter 2

GREEN-MARL by Example

In this chapter, we introduce the language with two example programs. These examples show most of the features of the language. We use them to informally introduce the type system and the read-write analysis before we present the formal specifications in the next chapters.

2.1 Closeness Centrality

Both examples (Figures 2.1 and 2.2) implement an algorithm to calculate the Closeness Centrality (CC) measure of every node in a graph. The centrality of a node in a graph is a measure that was first used in the social sciences by Bavalas [6]. The idea behind centrality was that the most central node needs the least amount of time to send a message to all other nodes in a graph. Since then many different variants of centrality have been proposed [19].

The CC value of a node in a graph is inversely related to the sum of the shortest paths to all other nodes in the graph. In other words, the reciprocal of the sum of the distance between the node and all other nodes in the graph:

$$CC(x) = \frac{1}{\sum_y d(x,y)}$$

where x and y are nodes and d is the distance function between two nodes.

To measure the distance between all pairs of nodes, there should be a path between all pairs. This property is called connectedness for undirected graphs, and strong connectedness for directed graphs [2].

The distance function measures the length of the shortest path between two nodes. In an unweighted graph this is the hop distance, i.e. the amount of edges in the path. Such a value is easy to find by doing a breadth-first search from one node until we come across the other node. In a weighted graph, a shortest path algorithm, e.g. Bellman-Ford [7,17], is needed to calculate the distance between two nodes.

2.2 Implementations

The two examples in Figures 2.1 and 2.2 calculate the CC measure for unweighted and weighted graphs respectively. We follow the above definition of Closeness Centrality

```
1  procedure ccOne(g: graph; cc: nodeProperty<double>) : bool {
2    if(g.numNodes() == 0) { // corner case: empty graph
3      return true; // we cannot pick a random node from an empty graph
4    }
5
6    // Kosaraju (simplified)
7    nodeProperty<bool> checked;
8    g.checked = false;
9    node t = g.pickRandom();
10   inDFS(n: g.nodes from t) {
11     n.checked = true;
12   }
13   if(any(v: g.nodes) {!v.checked}) {
14     return false; // Graph is not strongly connected
15   }
16   g.checked = false;
17   inDFS(n: g^.nodes from t) {
18     n.checked = true;
19   }
20   if(any(v: g.nodes) {!v.checked}) {
21     return false; // Graph is not strongly connected
22   }
23
24   // Closeness Centrality
25   foreach(n: g.nodes) {
26     long levelSum = 0;
27     inBFS(v: g.nodes from n) {
28       levelSum += currentBFSLevel();
29     }
30     n.cc = 1.0 / (double) levelSum;
31   }
32   return true;
33 }
```

Figure 2.1 – Closeness Centrality (Unit Length) – Simplified.

fairly closely in the implementation in GREEN-MARL.

GREEN-MARL works with directed graphs. These graphs carry only nodes and edges, an arbitrary ordering between nodes, an arbitrary ordering between edges, and the direction of each edge. A weighted graph is not a single entity in GREEN-MARL, it is instead modelled as a directed graph with a separate weight property defined for every edge.

Properties can be defined for nodes or edges of a graph. Since we calculate the CC value for every node in the graph, the return value of both implementations is a floating point number property on nodes (`cc : nodeProperty<double>`). However, the graph that is input may not be strongly connected, which would make the CC values for the nodes undefined. Therefore the examples have a Boolean return value indicating whether the graph is strongly connected, and an *output argument* to return the CC values.

Unweighted. In the first example, procedure `ccOne` in Figure 2.1, we define the input argument `g`, which is the unweighted graph, the output argument `cc` after the semicolon, which are the CC values, and the Boolean return type.

Between lines 7 and 22 we check the graph for strong connectedness. We do this by picking an arbitrary node and determining by depth-first search that it can access all other nodes (8-15). Then we do another depth-first search from the same node on the reverse graph to check that all nodes can reach this node (16-22).

The actual calculation of the measure is between lines 25 and 31. For the CC value of a node we use a breadth-first search to find a shortest path to other nodes. The hop distance is equal to the length of each path, we can sum up the current hop distance from the start node for every node we visit in the breadth-first search.

We use a Boolean node property `checked` to track accessible nodes (7). The property is initialised to **`false`** with a *group assignment* (8). This group assignment is syntactic sugar for traversing all nodes (or edges for an edge property) in the graph (or collection) and setting the property to **`false`** for each node. Moving on, we pick a random start node `t` in the graph with GREEN-MARL's built-in functions (9). The depth-first search **`inDFS`** traverses the nodes of the graph sequentially from the start node (10–12), and checks off visited nodes (11). Afterwards we see if any nodes were not reached (13). The **`any`** expression is a *reduction expression*, which reduces the values of a body expression to a single value. Next we reset the `checked` property, and do the depth-first search on the reverse graph `g^`, i.e. the same graph with the edge directions reversed. Again we see if we have reached all nodes, as this means that all nodes have a path to our start node (16–22).

For every node in parallel (25) we calculate the sum of all shortest paths (26). The breadth-first search (27–29) is also a parallel construct. It provides a built-in procedure for current level (28), which is the hop distance from the start node. The addition assignment `+=` is a numeric *reduction assignment*, which can be used to compute a sum in a parallel context without race conditions. Within parallel contexts `a += b` is *not* equal to `a = a + b`, the latter would result in a race. Finally, we take the reciprocal of the sum for the CC value (30).

Weighted. Procedure `ccVar` in Figure 2.2 calculates the CC values for all nodes on a weighted graph. The shortest path in a weighted graph is not necessarily the path with the least hops, therefore we switch from a breadth-first search to the Bellman-Ford algorithm [7,17]. The Bellman-Ford algorithm gradually minimises the length of the paths from infinity (10) towards the shortest path. It does this by iterating over all nodes to which a shorter path has been found in the last iteration (17). For each of these it checks if through this node its neighbours (18) can be reached with a shorter path (20). When the Bellman-Ford algorithm is done, any paths of infinite length indicate that the graph is not strongly connected. If none are infinite, the sum of the paths is used to calculate the CC value.

We calculate the CC value for every node sequentially (8). If we did this in parallel, there would be data races on the four properties that are defined at the start. The distance property `dist` is initialised at positive infinity, except for the `root` node (10). Positive infinity can be typed with any numeric type, not just the floating point types. We use a group assignment feature where the current node (or edge for edge properties) can be referenced on the right-hand side with **`_`**.

The `updated` property marks nodes that have been updated in the last iteration. Each iteration (16–28), the program goes over all these `updated` nodes (17), and considers its neighbours, the nodes connected through an outgoing edge, with the **`nbrs`**

```
1  procedure ccVar(g: graph, len: edgeProperty<double>;
2      cc: nodeProperty<double>) : bool {
3    nodeProperty<bool> updated;
4    nodeProperty<bool> updatedNext;
5    nodeProperty<double> dist;
6    nodeProperty<double> distNext;
7
8    for(root: g.nodes) {
9      // Bellman-Ford
10     g.dist    = (_ == root) ? 0 : +INF;
11     g.updated = (_ == root) ? true : false;
12     g.distNext    = _.dist;
13     g.updatedNext = _.updated;
14     bool notDone = true;
15
16     while(notDone) {
17       foreach(n: g.nodes)(n.updated) {
18         foreach(s: n.nbrs) {
19           edge e = s.toEdge();
20           s.distNext <s.updatedNext> min= n.dist + e.len <true>;
21         }
22       }
23
24       g.dist = _.distNext;
25       g.updated = _.updatedNext;
26       g.updatedNext = false;
27       notDone = any(n: g.nodes){n.updated};
28     }
29
30     // Closeness Centrality
31     bool b = any(v:g.nodes){v.dist == INF};
32     double pathSum = sum(v:g.nodes){v.dist};
33
34     if(b) { // disconnected graph
35       return false;
36     } else {
37       root.cc = 1.0 / pathSum;
38     }
39   }
40
41   return true;
42 }
```

Figure 2.2 – Closeness Centrality (Double Length) – Simplified.

range (18). If a new shortest path is found by going from the updated node to the neighbour node, the `distNext` is updated to the new shortest path length (20). This is again done by a reduction assignment to make it safe in the parallel context. With comparison reduction assignments, extra arguments can be given in angled brackets, as we do with `updatedNext`. This allows the arguments to be updated atomically when a new minimum is found. After the nested parallel loops, the `distNext` and `updatedNext` properties are copied to their counterparts (24–25) and `updatedNext` is reset (26). When no nodes were updated, the algorithm is done (27).

After the Bellman-Ford algorithm has finished, we check for an infinite length shortest path with another **`any`** reduction (31). If we find an infinite length (34), we return early (35) since the graph is not strongly connected. Otherwise we again take the reciprocal of the sum of paths (32) as the CC value (37).

The sum is calculated outside of the if statement to expose more optimisation opportunities to the compiler. We can reason that the sum is only one time extra if we move it out of the else clause, because the other branch of the if statement holds a return statement. And in that case the procedure stops early anyway, so this is ok. The current compiler cannot reason about this and will not move calculations outside conditional statements. One of the optimisation opportunities we expose to the compiler by moving the sum out of the if statement is a fusion of the **`any`** and **`sum`** reductions into a single loop over the nodes of the graph.

2.3 Type system

The examples illustrate the use of GREEN-MARL and some of its types. Among those are both general purpose types and domain-specific types. In this section, we give an informal overview of those types as well as the types of functions and procedures. We provide a formal treatment of the type system in the next chapter.

General purpose types. GREEN-MARL provides standard numeric types **`int`**, **`long`**, **`float`** and **`double`**, the standard **`bool`** and **`string`** types, and the **`date`** type. The **`date`** type has no unique literals, instead all string literals can be typed as **`date`**. Built-in procedures and functions can set the pattern that instructs how a date is parsed from such a string literal, and such patterns can also be used to parse a string into a date at run time.

Domain specific types. Beside general purpose types, GREEN-MARL provides the **`graph`** type for directed graphs, and the **`node`** and **`edge`** types for graph elements. These domain specific types are parametrised with the name of the graph they belong to in round brackets, e.g. **`node`**`(g)`. This graph parameter was left implicit in the examples, as it can be inferred when only one graph is in scope. To use a graph or graph element, the usual way as seen in the examples is to traverse a range such as the nodes of the graph `g.`**`nodes`** or the neighbours of a node `n.`**`nbrs`**.

These graph elements can also be collected in sets, sequences, and orders (sequences with unique elements): **`nodeSet`**, **`edgeSet`**, **`nodeSequence`**, **`nodeSequence`**, **`nodeOrder`** and **`edgeOrder`**. These types are again parametrised with a graph name. A collection can be traversed by its **`items`** range.

Local declarations. Local variables of the above mentioned types all belong to the namespace of variables. For these local variables, shadowing or overloading is disallowed.

Graph properties. Although defined in a similar fashion, names with a property type reside in a different namespace, and are accessed by a dot notation: `n.prop`. Another feature of properties is that they may be overloaded on the graph that they belong to. This allows a property with the same name, e.g. `weight`, to be defined on edges of two different graphs in the same procedure. Property types are not only parametrised by graph name, they are also parametrised by the type that they hold. Type parameters in GREEN-MARL are provided between angular brackets <>. The graph parameter, if specified, comes after the type parameters.

Top level declarations. Procedures in GREEN-MARL are top-level constructs with a separate namespace. They optionally have input arguments, output arguments, and/or a return value. Input arguments are read-only values and are the only way to introduce a graph. Output arguments are for returning extra values, but may also be initialised by the caller to provide more input. User-defined procedures cannot be parametrised by type, but they can take graphs as input arguments, which results in a procedure type that is polymorphic in graph names. User-defined procedures must not be overloaded. They cannot be defined anywhere other than at the top level, therefore shadowing is impossible.

Built-in functions and procedures. Built-in functions like `pickRandom` in `ccOne` are called on a subject with a dot notation, and can take more (input) arguments in their round brackets. Functions types can be parametrised by graph names, types, or both. `pickRandom` takes a graph, and returns a random node *of that graph*. The type has to be parametrised by graph name, and the graph type need to be embellished with its name itself to show this connection. This results in the following type for `pickRandom`: $\forall n.$ `F<graph(`n`),⟨⟩,N(`n`)>`. The type describes that for all names n, this is a function that takes a graph with the name n as a subject, it takes an empty list of arguments, and has a node of that same graph n as a result.

Most of the functions in the API provide a way to manipulate collections. Beside built-in functions, some procedures are also provided as built-ins. The functionality these procedure provide include mathematical operations and string to date conversion. These built-in procedures can be overloaded.

2.4 Dependence analysis

GREEN-MARL provides both sequential and parallel loops. Beside explicit parallel loops, the language provides higher-level abstractions and syntactic sugar for implicitly parallel loops. For example, the breadth-first search is a parallel domain abstraction, reduction expressions and group assignments are syntactic sugar for parallel loops.

Parallel constructs can cause data races. GREEN-MARL employs a dependence analysis called the read-write analysis to check for data-races, check for invariants of

race mitigation constructs, and inform optimisations [23]. The analysis gathers information about which names are written and read in which parts of the program. With this read-write information we can derive the data dependence between statements.

The read-write analysis is an intraprocedural, bottom-up analysis. During the analysis, we collect two pieces of information for each local declaration: modes of access and the property access patterns. The possible access modes of access are: **read**, **write**, **defer**(·) and **reduce**(·,·). The read and write modes are the normal modes that can cause data races in parallel contexts. The defer and reduce modes have specific invariants that need to be adhered.

The information on how properties are accessed is a refinement of the analysis that can identify more situations as safe from data races. It can also indicate that loops can be merged despite a dependence between the two loops. The possible patterns are a single access to name n, access to a **unique** set of names, and access to a **random** sample of possibly overlapping names.

The bottom-up analysis manifests in itself in the abstraction over the effect of a statement with sub-statements. Every statement can be described in terms of its read-write information. For statements comprised of sub-statements, the read-write information is limited to the outside observable effects of the statement when executed fully. For example, a block of statements with a local declaration inside will not have that local declaration in its read-write information.

Breadth-first search. For the first example of an analysis, we use the breadth-first search of `cc0ne`. To avoid data races in this parallel breadth-first search, the `levelSum` is summed with a reduction assignment:

```
27  inBFS(v: g.nodes from n) {
28    levelSum += currentBFSLevel();
29  }
```

The analysis is bottom-up and first considers the breadth-first search body. The body is a reduction **reduce**(·,·), which is scoped by the breadth-first search `v`, and based on an addition `+=`.

line	target	rw mode
28	`levelSum`	**reduce**(v,+=)

At the level of the breadth-first search itself, the outside effect of the search is only a write to `levelSum`, a read on the start node `n`, and the graph `g`:

line	target	rw mode
27-29	`levelSum`	**write**
	`n`	**read**
	`g`	**read**

To illustrate the detection of a conflict, we can change the reduction assignment in the breadth-first search into a normal assignment and addition:

```
27  inBFS(v: g.nodes from n) {
28    levelSum = levelSum + currentBFSLevel();
29  }
```

This changes the analysis results for the breadth-first search body to a normal read and write on `levelSum`.

line	target	rw mode
28	`levelSum`	**`write`**
	`levelSum`	**`read`**

On the breadth-first search level, a normal write is now encountered in the body statement. Since these write are done in parallel, this is a **`write-write`** conflict. Beside this conflict, there is also a **`read-write`** conflict, as the read and write in the body statement are not guaranteed to happen atomically.

line	target	rw mode
27-29	`levelSum`	**`write-write`** conflict
	`levelSum`	**`read-write`** conflict
	`n`	**`read`**
	`g`	**`read`**

Nested parallel loops. The scope of a reduction assignment is automatically determined. The context that scopes the reduction is the outermost parallel context where the target of the reduction is still defined. The breadth-first search in the previous excerpt is inside of a parallel loop:

```
25  foreach(n: g.nodes) {
26    long levelSum = 0;
27    inBFS(v: g.nodes from n) {
28      levelSum += currentBFSLevel();
29    }
30    n.cc = 1.0 / (double) levelSum;
31  }
```

However, the `levelSum` is defined inside of that loop, therefore the loop cannot be the scope of the reduction, since that would put the write to the variable at the point where it goes out of scope.

Consider instead the nested loops from the `ccVar` procedure:

```
17  foreach(n: g.nodes)(n.updated) {
18    foreach(s: n.nbrs) {
19      edge e = s.toEdge();
20      s.distNext <s.updatedNext> min= n.dist + e.len <true>;
21    }
22  }
```

In this case the reduction target is the `distNext` node property. It is a minimising reduction, which can come with extra arguments, which are considered part of the reduction (**`reduce`**(·,**`arg`**(·))).

line	target	rw mode
20	`distNext`	**`reduce`**(`n`,**`min`**)
	`updatedNext`	**`reduce`**(`n`,**`arg`**(**`min`**))

At the level of the inner loop the iterator through which is the properties are accessed goes out of scope, but the analysis results stay the same, because `n` is still in scope.

line	target	rw mode
18-21	`distNext`	**`reduce`**(`n`,**`min`**)
	`updatedNext`	**`reduce`**(`n`,**`arg`**(**`min`**))

Next is the outer loop, which scopes the reduction. Here `n` goes out of scope, the reduction ends, and the observable effect becomes a write.

line	target	rw mode
17-22	`distNext`	**`write`**
	`updatedNext`	**`write`**

If the scope of the reduction was set to the inner loop, then the reduction would change to a write one level earlier. At the level of the outer loop, the write would be encountered in the body, and a **`write-write`** conflict would be flagged. So reduction assignments are scoped by the outermost parallel context for a reason. The read-write analysis results for the reduced properties shows that the outer scope that the language chooses avoids data races.

Access patterns. So far, we have purposely ignored property access patterns. In the following example we show the access patterns of the last excerpt.

```
17  foreach(n: g.nodes)(n.updated) {
18    foreach(s: n.nbrs) {
19      edge e = s.toEdge();
20      s.distNext <s.updatedNext> min= n.dist + e.len <true>;
21    }
22  }
```

line	target	rw mode	access pattern
20	`distNext`	**`reduce`**`(n,`**`min`**`)`	`s`
	`updatedNext`	**`reduce`**`(n,`**`arg`**`(`**`min`**`))`	`s`

The simplest accessor of a property is a single value, in the form of a local variable, as is the case for the `distNext` and `updatedNext` properties of the last excerpt. When the variable `s` goes out of scope, we must abstract over the access pattern. A normal local variable can have any value from any kind of expression. We do not try to predict that value as it can be truly random (e.g. `g.pickRandom()`). Instead we use the access pattern **`random`**, which is not fit for optimisations. In the excerpt, the accessor is an iterator. An iterator over neighbours visits each node only once, therefore we use the access pattern **`unique`**.

line	target	rw mode	access pattern
18-21	`distNext`	**`reduce`**`(n,`**`min`**`)`	**`unique`**
	`updatedNext`	**`reduce`**`(n,`**`arg`**`(`**`min`**`))`	**`unique`**

If a property is accessed in a **`random`** pattern, it stays random when repeated. The **`unique`** pattern comes from a repetition. Repeating that repetition is likely to result in multiple property accesses in the same place. Therefore the **`unique`** pattern is also turned into a **`random`** pattern when found in the body of a parallel loop:

line	target	rw mode	access pattern
17-22	`distNext`	**`write`**	**`random`**
	`updatedNext`	**`write`**	**`random`**

Loop merging. In the following excerpt we have four short lines that initialise some properties in `ccVar`.

```
10  g.dist    = (_ == root) ? 0 : +INF;
11  g.updated = (_ == root) ? true : false;
12  g.distNext    = _.dist;
13  g.updatedNext = _.updated;
```

Each of these assignments is a group assignment, which desugars into a parallel loop over the nodes of the graph. The desugared version is given below. The assignment uses the loop iterator instead of the graph on the left-hand side and instead of the underscore on the right-hand side.

```
A10  foreach(n1: g.nodes) {
A11    n1.dist  = (n1 == root) ? 0 : +INF;
A12  }
A13  foreach(n2: g.nodes) {
A14    n2.updated  = (n2 == root) ? true : false;
A15  }
A16  foreach(n3: g.nodes) {
A17    n3.distNext  = n3.dist;
A18  }
A19  foreach(n4: g.nodes) {
A20    n4.updatedNext  = n4.updated;
A21  }
```

The read-write analysis is applied to the desugared program. The results for the first loop body are that `dist` is written through `n1`, `n1` is read, and `root` is read. These resolve into unique writes on `dist`, and a read on `root` for the entire loop.

line	target	rw mode	access pattern
A11	dist	**write**	n1
	n1	**read**	
	root	**read**	
A10-A12	dist	**write**	**unique**
	root	**read**	

The other analysis results looks similar:

line	target	rw mode	access pattern
A10-A12	dist	**write**	**unique**
	root	**read**	
A13-A15	updated	**write**	**unique**
	root	**read**	
A16-A18	distNext	**write**	**unique**
	dist	**read**	**unique**
A19-A21	updatedNext	**write**	**unique**
	updated	**read**	**unique**

As a result, the only dependences between these loops are on `dist` and `updated`, which are written, and then read. Because all the ranges are equal and all the dependences are **`unique`**, the compiler can merge the loops:

```
foreach(n: g.nodes) {
  n.dist    = (n == root) ? 0 : +INF;
  n.updated = (n == root) ? true : false;
  n.distNext    = n.dist;
```

```
    n.updatedNext = n.updated;
  }
```

A final detail to point out is the conditional expression that the first two group assignments use. It may seems strange to do a check to see if the iterator is equal to the root node, since this only happens once. But if the `dist` property were assigned `+INF` over the entire range, and `root.dist` was assigned `0` afterwards, that second assignment would be in the way of merging all the loops into one.

```
B10  g.dist    = +INF;    // changed
B11  root.dist = 0;       // new
B12  g.updated = false;   // changed
B13  root.updated = true; // new
B14  g.distNext    = _.dist;
B15  g.updatedNext = _.updated;
```

According to the dependences the assignments to `root` cannot be moved out of the way, as the **`write`** to `dist` has to be read by the `distNext` group assignment. Therefore we can only merge two pairs of loops:

```
C10  foreach(n: g.nodes) {
C11    n.dist    = +INF;
C12    n.updated = false;
C13  }
C14  root.dist = 0;
C15  root.updated = true;
C16  foreach(n: g.nodes) {
C17    n.distNext    = n.dist;
C18    n.updatedNext = n.updated;
C19  }
```

Chapter 3

Type System

In the last chapter, we described GREEN-MARL by example. In this chapter we present the formal specification of GREEN-MARL's type system. The type system of a programming language is typically described in terms of well-typedness and well-formedness judgements for the syntactic domains of a language. In Figure 3.1 we present the central judgements, syntax and semantic domains of the specification. The judgements are:

- *Well-formed units:* $\vdash u$
  Well-formed procedure declarations: $\Gamma \vdash p$
  This is the starting point of the judgements. A well-formed compilation unit consists of zero or more well-formed procedures. The well-formedness of procedures is judged under environment Γ.

  For these environments we use lookup notation $\Gamma(n)$ and update notation $\Gamma[\, n \mapsto \tau\,]$. Under the hood we model environments as point-wise defined functions. A point-wise defined function $f : X \rightarrow_{fin} Y$ is only defined for a finite subset of domain X. For a value $x \in X$ outside of the defined subset $f(x) = \bot$. The entirely undefined function is denoted by $\bot$.

  The environment Γ we use is composed of sub-environments for procedures Γ_p, graph properties Γ_g and local variables Γ_v. In the rules we do not explicitly deconstruct the Γ, instead we use the subscript of the sub-environment when we do a lookup or update.

- *Procedure declarations:* $\Gamma \vdash p : \Gamma'$
  Procedures are named and contribute to the environment. The procedure declaration are extracted and added to the environment separately. Only then are the procedures judged on well-formedness, to allow all procedures to refer to each other.

- *Well-formed statements:* $\tau, \gamma, \Gamma \vdash s$
  Statements are judged well-formed under environment Γ, return type τ and formal graph arguments γ. The return type of a procedure is propagated downward to type-check the return statement. The formal graph arguments to a procedure are also propagated from the procedure level, for use in the semantic type translation.

We follow ideas from Implicit Propagation in Structural Operational Semantics [31] to reduce notational overhead in many of the rules. In particular, we leave off auxiliary arguments that are not used in a rule. These are considered implicitly propagated. We print such auxiliary arguments in a grey box in the judgement for clarity.

- *Variable and graph property declarations:* $\tau, \gamma, \Gamma \vdash s : \Gamma'$
  Some statements can introduce local names that are only visible within a list of statements after the declaration statement. Most statements are simpler and have a well-formedness judgement. For these we have a single rule that passes the environment on unchanged.

- *Well-typed expressions:* $\Gamma \vdash e : \tau$
  We use semantic types τ instead of syntactic types t for our judgements, because not all types have a syntactic form (e.g. `void`) and other types can have partially inferred information (e.g. `graph`(n)).

- *Semantic type translation:* $\gamma \vdash t \Rightarrow \tau$
  Translation of the syntactic types to semantic ones is done with a translation judgement that takes a list of the formal graph arguments of the current procedure. This is used to validate or infer the graph argument to some graph-related types.

In the remainder of the chapter we specify the rules of the judgements based on the structure of the syntactic types. These are graph property types t_g and value types t_v, where t_v is further subdivided into primitive types t_p, graphs and graph elements `graph`, t_e, collection types t_c, t_{cc}, and map types t_m.

3.1 Variable Declarations and References

Names in GREEN-MARL can refer to functions, procedures, graph properties, and variables. Each name is used in a syntactically distinct way throughout a program. Figure 3.2 describes the use of names for local variables in declarations, assignments, references, and the variable environment.

Local declarations. The names of graph properties and local variables are introduced with a local declaration. We only handle the local variables [decl-v] in Figure 3.3, since graph properties are in a different sub-environment. The value type t_v is translated to semantic type τ with the semantic type translation judgement. Note that there is a check that the name is not defined yet, which purposely rules out shadowing.

References. A reference can be used on the left-hand side of an assignment [assign] or on the right-hand side in an expression [ref]. Some variables in GREEN-MARL can be read-only, therefore we describe this write or read context with α. With an auxiliary judgement for references we judge that reading a reference [ref-r] can always be done, whereas writing to a reference [ref-w] requires the reference to be writable.

syntax

n		procedure, function, variable, property names	Section 3.1
t	$= t_v$	value types	
	$\mid t_g$	graph property types	Section 3.5
t_v	$= t_p$	primitive types	Section 3.2
	$\mid$ **graph** $\mid t_e$	graph type and graph element types	Section 3.3
	$\mid t_c \mid t_{cc}$	collection types	Section 3.4
	$\mid t_m$	map types	Section 3.6
e		expressions	
s		statements	
p		procedure declarations	Section 3.7
u	$= p^*$	compilation units	Section 3.7

semantic domains

γ	$= n^*$	formal graph arguments	Sections 3.1, 3.7
Γ	$= \Gamma_v$	variable environment	Section 3.1
	$\times \Gamma_g$	graph property environment	Section 3.5
	$\times \Gamma_p$	procedure environment	Section 3.7
τ	$= \tau_p$		
	$\mid$ **graph**$(n) \mid \tau_e$		
	$\mid \tau_c \mid \tau_{cc}$		
	$\mid \tau_m$		
	$\mid \tau_g$		
	$\mid \tau_i$	iterator types	Sections 3.3, 3.4
	$\mid$ **void**	void type	Section 3.7

semantic judgements

$\vdash u$	well-formed units	Section 3.7
$\Gamma \vdash p : \Gamma'$	procedure declarations	Section 3.7
$\Gamma \vdash p$	well-formed procedure declarations	Section 3.7
$\tau, \gamma, \Gamma \vdash s : \Gamma'$	variable and graph property declarations	Sections 3.1, 3.5, 3.6
$\tau, \gamma, \Gamma \vdash s$	well-formed statements	
$\Gamma \vdash e : \tau$	well-typed expressions	
$\gamma \vdash t \Rightarrow \tau$	semantic type translation	

Figure 3.1 – GREEN-MARL's central syntactic domains, semantic domains and judgements, along with the sections where they are defined.

syntax

s	$=$	$t\ n;$	local declaration
	$\mid$	$e_r = e;$	assignments
	$\mid$	$\{s^*\}$	blocks
	$\mid$	$\ldots$	
e	$=$	e_r	references
	$\mid$	$\ldots$	
e_r	$=$	n	
	$\mid$	$\ldots$	

semantic domains

Γ_v	$=$	$n \rightarrow_{fin} (\tau \times \alpha)$	variable environment
α	$=$	$\mathbf{r} \mid \mathbf{w}$	access context

semantic judgements

$\alpha, \Gamma \vdash e_r : \tau$ references

Figure 3.2 – Names, local variables and related syntax.

local declarations $\boxed{\tau, \gamma, \Gamma \vdash s : \Gamma'}$

$$\frac{\gamma \vdash t_v \Rightarrow \tau \ \wedge\ \Gamma_v(n) = \bot \ \wedge\ \Gamma_v[n \mapsto \langle \tau, \mathbf{w} \rangle] = \Gamma'}{\tau', \gamma, \Gamma \vdash t_v\ n; : \Gamma'}\ [\text{decl-v}] \qquad \frac{\tau', \Gamma \vdash s}{\tau', \gamma, \Gamma \vdash s : \Gamma}\ [\text{non-decl}]$$

well-typed references $\boxed{\alpha, \Gamma \vdash e_r : \tau}$

$$\frac{\Gamma_v(n) = \langle \tau, \_ \rangle}{\mathbf{r}, \Gamma \vdash n : \tau}\ [\text{ref-r}] \qquad \frac{\Gamma_v(n) = \langle \tau, \mathbf{w} \rangle}{\mathbf{w}, \Gamma \vdash n : \tau}\ [\text{ref-w}]$$

well-typed expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\mathbf{r}, \Gamma \vdash e_r : \tau}{\Gamma \vdash e_r : \tau}\ [\text{ref}]$$

well-formed statements $\boxed{\tau, \Gamma \vdash s}$

$$\frac{\mathbf{w}, \Gamma \vdash e_r : \tau \ \wedge\ \Gamma \vdash e : \tau}{\Gamma \vdash e_r = e;}\ [\text{assign}] \qquad \frac{\Gamma \vdash^{\twoheadrightarrow} s^* : \_}{\Gamma \vdash \{s^*\}}\ [\text{block}]$$

Figure 3.3 – Basic name rules.

Blocks. Any name introduced inside a block is only visible within that block [block]. The resulting environment of s^* is ignored (_). The reverse connection between the two statement judgements is in [non-decl], where well-formed non-declaration statements preserve the environment unchanged.

The $\vdash^{\twoheadrightarrow}$ in the block rule is a short-hand for fold from the left. The expanded rule is:

$$\frac{\tau,\Gamma \vdash s_1 : \Gamma_1 \quad \wedge \quad \ldots \quad \wedge \quad \tau,\Gamma_{n-1} \vdash s_n : \Gamma_n}{\tau,\Gamma \vdash \{\langle s_1,\ldots,s_n\rangle\}} \qquad \text{[block-expanded]}$$

3.2 Primitive Types

GREEN-MARL has numeric, boolean, string and date types. Figure 3.4 summarises their related expressions and their related statements. The figure also shows how the type translation of primitive types is trivial.

Numeric Types. GREEN-MARL supports the typical numeric types `int`, `long`, `float` and `double`, but there are only literals for `int` and `float`. The `long` and `double` literals are missing. (see Figure 3.5)

GREEN-MARL follows IEEE 754 [1] and provides numeric literals for positive and negative infinity `INF`. The floating point types have special infinity values, but the integer types do not normally have such special values. GREEN-MARL defines the extreme values -2^{31} and $2^{31}-1$ as the infinities for `int` and similarly -2^{63} and $2^{63}-1$ for `long`. These infinity values do not have defined behaviour for arithmetic, but they are not statically excluded from arithmetic operations.

Numeric expressions include the standard arithmetic operations [num-op], [umin] and [abs]. Comparison and equality operators are also supported [n-cop].

Numeric types can be explicitly cast using a C-like cast syntax. Note that this operation is restricted to numeric casts, it is not a general escape hatch in the type system. The numeric types can also be implicitly coerced to types with larger ranges with [il-coerce], [lf-coerce] and [fd-coerce]. We treat the infinity literals as `int` only since these implicit coercion take them to the other types.

String and Date Types. Strings and dates are introduced with the same string literal. In the case of a date, the string literal's content is interpreted as a date. Strings and dates can be compared and checked for equality [s-cop] [d-cop].

Boolean Type. GREEN-MARL offers standard booleans literals `true` and `false`. The standard have logic operations are available, as well as an if-else expression by means of the C-style ternary operator. Booleans are also equatable [eop], but do not have comparison defined.

The other primitive types have an order and can be compared. The comparison operator domain o_c contains the equality operators o_e, so the string, date and numeric types have equality operations defined as well.

Finally if, if-else, while, and do-while are the standard conditional statements of GREEN-MARL.

syntax

t_p	=	t_n	numeric types
	|	**string** | **date**	string and date types
	|	**bool**	boolean type
t_n	=	**int** | **long** | **float** | **double**	
e	=	+**INF** | −**INF** | l_i | l_f	numeric literals
	|	l_s	string and date literals
	|	**true** | **false**	boolean literals
	|	$-e$ | $\|e\|$ | $e\ o_n\ e$ | $(t_p)\ e$	numeric expressions
	|	$e\ o_c\ e$ | $e\ o_e\ e$	boolean expressions
	|	$!e$ | $e\ o_l\ e$ | e ? e : e	
	|	...	
s	=	**if**(e) s **else** s	conditional statements
	|	**if**(e) s	
	|	**while**(e) s	
	|	**do** s **while**(e);	
	|	...	
o_n	=	+ | − | * | / | %	numeric operators
o_c	=	< | <= | >= | > | o_e	comparison operators
o_e	=	== | !=	equality operators
o_l	=	&& | ||	logic operators
l_i			integer literals
l_f			floating point literals
l_s			string and date literals

semantic domains

τ_p	=	t_p
τ_n	=	t_n

semantic type translation $\boxed{\gamma \vdash t \Rightarrow \tau}$

$\gamma \vdash t_p \Rightarrow t_p$ [sem-pt]

Figure 3.4 – Primitive types, related syntax, and semantic type translation.

well-typed expressions - numeric types $\boxed{\Gamma \vdash e : \tau}$

$$\texttt{+INF} : \texttt{int} \quad \text{[p-inf]} \qquad \texttt{-INF} : \texttt{int} \quad \text{[n-inf]}$$

$$l_i : \texttt{int} \quad \text{[i-lit]} \qquad l_f : \texttt{float} \quad \text{[f-lit]}$$

$$\frac{e : \tau_n}{-e : \tau_n} \quad \text{[umin]} \qquad \frac{e : \tau_n}{|e| : \tau_n} \quad \text{[abs]}$$

$$\frac{e_1 : \tau_n \ \wedge \ e_2 : \tau_n}{e_1 \ o_n \ e_2 : \tau_n} \quad \text{[num-op]} \qquad \frac{e_1 : \tau_n \ \wedge \ e_2 : \tau_n}{e_1 \ o_c \ e_2 : \texttt{bool}} \quad \text{[n-cop]}$$

$$\frac{e : t_n}{(t_n) \ e : t_n} \quad \text{[cast]} \qquad \frac{e : \texttt{int}}{e : \texttt{long}} \quad \text{[il-coerce]}$$

$$\frac{e : \texttt{long}}{e : \texttt{float}} \quad \text{[lf-coerce]} \qquad \frac{e : \texttt{float}}{e : \texttt{double}} \quad \text{[fd-coerce]}$$

well-typed expressions - string and date types

$$l_s : \texttt{string} \quad \text{[s-lit]} \qquad l_s : \texttt{date} \quad \text{[d-lit]}$$

$$\frac{\begin{array}{l} e_1 : \texttt{string} \\ \wedge \quad e_2 : \texttt{string} \end{array}}{e_1 \ o_c \ e_2 : \texttt{bool}} \quad \text{[s-cop]} \qquad \frac{\begin{array}{l} e_1 : \texttt{date} \\ \wedge \quad e_2 : \texttt{date} \end{array}}{e_1 \ o_c \ e_2 : \texttt{bool}} \quad \text{[d-cop]}$$

well-typed expressions - boolean types

$$\texttt{true} : \texttt{bool} \quad \text{[true]} \qquad \texttt{false} : \texttt{bool} \quad \text{[false]}$$

$$\frac{e : \texttt{bool}}{!\, e : \texttt{bool}} \quad \text{[neg]} \qquad \frac{e_1 : \texttt{bool} \ \wedge \ e_2 : \texttt{bool}}{e_1 \ o_l \ e_2 : \texttt{bool}} \quad \text{[lop]}$$

$$\frac{\begin{array}{l} e_1 : \texttt{bool} \\ \wedge \quad e_2 : \texttt{bool} \end{array}}{e_1 \ o_e \ e_2 : \texttt{bool}} \quad \text{[eop]} \qquad \frac{\begin{array}{l} e_1 : \texttt{bool} \\ \wedge \ e_2 : \tau \ \wedge \ e_3 : \tau \end{array}}{e_1 \ ? \ e_2 \ : \ e_3 : \tau} \quad \text{[ter-if]}$$

well-formed statements $\boxed{\tau, \gamma, \Gamma \vdash s}$

$$\frac{e : \texttt{bool} \ \wedge \ \vdash s_1 \ \wedge \ \vdash s_2}{\vdash \texttt{if}(e) \ s_1 \ \texttt{else} \ s_2} \quad \text{[if-else]} \qquad \frac{e : \texttt{bool} \ \wedge \ \vdash s_1}{\vdash \texttt{if}(e) \ s_1} \quad \text{[if]}$$

$$\frac{e : \texttt{bool} \ \wedge \ \vdash s}{\vdash \texttt{while}(e) \ s} \quad \text{[while]} \qquad \frac{e : \texttt{bool} \ \wedge \ \vdash s}{\vdash \texttt{do} \ s \ \texttt{while}(e);} \quad \text{[do]}$$

Figure 3.5 – Primitive type related literals, expressions and statements.

syntax

t_e	=	**N** | **N**(n)	nodes
	|	**E** | **E**(n)	edges
s	=	...	
	|	**for** i s	sequential loops
	|	**foreach** i s	parallel loops
	|	**inDFS** i_s s **inPost**(e) s	depth-first searches
	|	**inBFS** i_s s **inReverse**(e) s	breadth-first searches
	|	e_r <= e | e_r <= e @ n	deferred assignments
	|	e_r ra_n e | e_r ra_l e | e_r <e_r^*> ra_c e <e^*>	reductions
	|	...	
e	=	...	
	|	**NIL**	node/edge literal
	|	ro_n i {e} | ro_l i {e}	reductions
	|	...	
ra_n	=	+= | *= | ra_c	numeric reduction assigments
ra_c	=	**max**= | **min**=	comparison reduction assigments
ra_l	=	&= | |=	logic reduction assigments
ro_l	=	**any** | **all**	logic reduction operators
i	=	(n: r)(e)	loop iterators
i_s	=	(n: n.**nodes from** n)(e)[e]	
	|	(n: n^.**nodes from** n)(e)[e]	search iterators
r	=	n.**nodes** | n.**edges** | n^.**edges**	graph ranges
	|	n.**inNbrs** | n.**outNbrs**	node ranges
	|	n.**inEdges** | n.**outEdges**	
	|	n.**upNbrs** | n.**downNbrs**	breadth-first search iterator ranges
	|	n.**upEdges** | n.**downEdges**	
	|	...	
ro_n	=	**sum** | **product** | **max** | **min**	numeric reduction operators

semantic domains

τ_e	=	**N**(n) | **E**(n)	graph elements
τ_i	=	**I**<ι, τ_e>	iterators
ι	=	**s**	standard iterator context
	|	**b**	breadth-first search context
	|	**n**	neighbour iteration context
	|	...	

semantic judgements

$\iota, \Gamma \vdash i_s : \Gamma'$ search iterators

$\Gamma \vdash i : \Gamma'$ loop iterators

$\Gamma \vdash r : \tau$ well-typed ranges

Figure 3.6 – Graph, nodes, edges and related syntax.

3.3 Graphs, nodes and edges

GREEN-MARL is a domain specific, graph oriented language, so it will come as no surprise that there is a built-in notion of graphs. Figure 3.6 introduces the graph type, and graph element types nodes and edges, along with traversals, iterators and ranges that relate to these types.

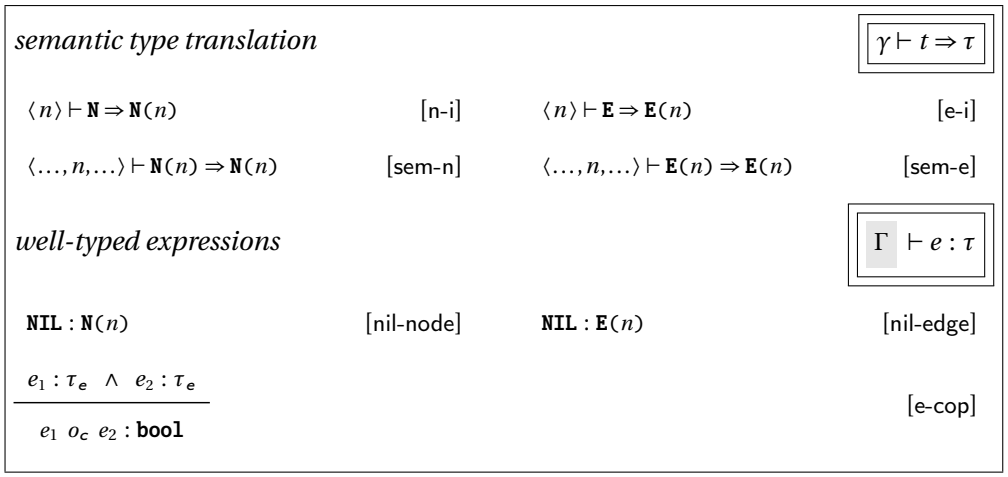


Figure 3.7 – Type translation and iterator type coercion.

Graphs. The `graph` type in GREEN-MARL may look like a normal type but it is not. First, there is no semantic type translation defined for the graph type, so its definition can be restricted. Section 3.7 defines this rule, where a graph-typed variable may only be introduced in the in-arguments of a procedure and is therefore read-only. The list of graphs in scope γ is also built at that level and passed down only to be read.

Second, a graph is not just a value, it is also a type parameter. A graph name is present within round brackets for any graph related type, to bind the type to that particular graph. For example, this allows the user to define a node of graph `g` as `N(g)`. When only one graph is in scope, the user can leave the graph parameter implicit. One of the reasons for having semantic types is the explication of graph parameters ([n-i] and [e-i] in Figure 3.7).

Graph elements. . Nodes and edges are nullable types through the `NIL` literal. Because `NIL` does not belong to a particular graph, the rules uses an arbitrary name n. The value itself is also not clearly a node or an edge, therefore [nil-node] and [nil-edge] both apply to the literal. Side-conditions of rules in a type derivation tree can eliminate the rule that is not applicable.

Nodes and edges have an arbitrary order, which allows a GREEN-MARL user to use the comparison operators on nodes, and on edges (see [e-cop]).

Breadth- and depth-first searches. GREEN-MARL offers breadth-first search [bfs] and depth-first search [dfs] (Figure 3.8). These searches go over a specified graph from a specified start node, and execute statements on the way forward and on the way back. The way back is described with `inReverse` and `inPost` respectively. Both statements for forward and reverse have access to the *iterator* of the search, which provides the current node of the search [it-coerce] and can be used in iterator-specific places.

The search follows the edges of the graph from the start node, but it can skip nodes that match a filter expression. These filter expressions are given in round brackets, one for the forward part, one for the backward part of the search. The

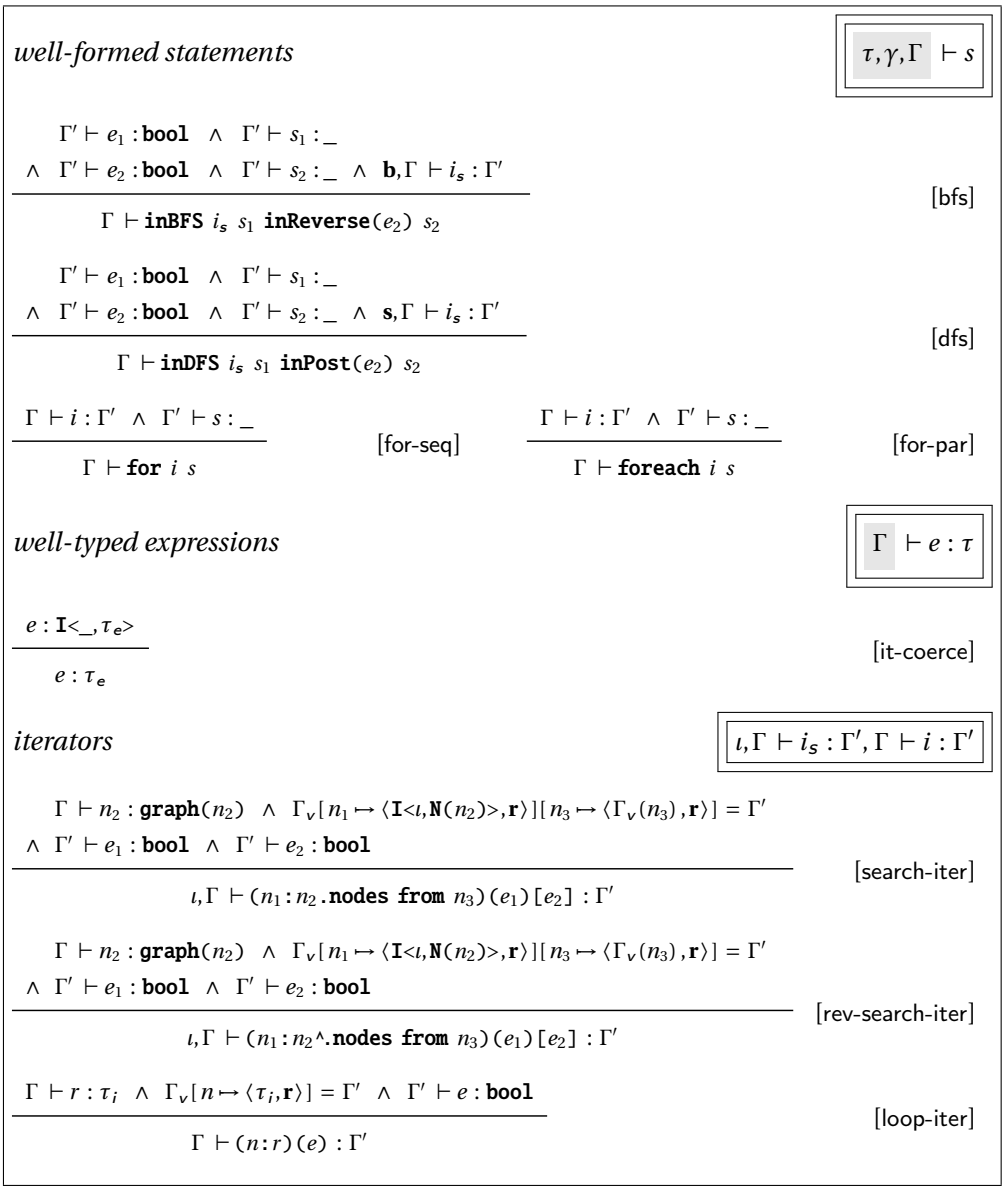


Figure 3.8 – Loops and traversals, iterators and comparison expressions.

search can be influenced more strongly by a navigator expression, which is given in square brackets to distinguish it from the filter expression. When a node matches the navigator expression, its edges are excluded from the search. The matched node is processed, but its unvisited neighbours are not unless there is another path to them.

The iterator name, graph, start node, forward filter, and navigator together make up the search iterator i_s. When the graph name is followed by a ^, the search is done over the reverse graph.

Sequential and parallel for loops. Iterators are also used GREEN-MARL's for loops. These are the sequential **`for`** and parallel **`foreach`**. Each loop has an iterator, a subject and range, and a filter expression. Ranges over graph-typed subjects are over all nodes or edges ([nodes] and [edges] in Figure 3.9). Again the ^ reverses the edges [rev-edges]. A node-typed subject gives rise to ranges over the neighbourhood of the node, based on in [in-nbrs][in-edges] or outgoing[1] edges, or on the direction of a breadth-first search. The breadth-first search direction up [up-nbrs][up-edges] is towards the start node and down[1] is towards unvisited nodes. Such a direction only makes sense for a breadth-first search and is therefore only possible on an iterator from a breadth-first search.

Deferred and Reduction Assignments. There are a number of special assignments available for parallel traversals. The deferred assignment [def-n] (Figure 3.10) consists of left-hand and right-hand side and a *bound*. This bound is an iterator that denotes the traversal wherein the assignment is deferred. After the execution of the traversal the assignment is visible, whereas within the traversal the old value is observed. When no bound is provided [def], it is inferred as the closest parallel traversal.

There are multiple flavours of reduction assignments [red-l][red-n] based on different operations ra_l and ra_n. These reduction assignments do not specify a bound, it is always inferred as the closest parallel traversal. The semantics of this reduction assignment is that after the traversal the original value and all the values that the traversal supplied are reduced and saved in the variable on the left-hand side. For example, if a **`foreach`** loop uses a `+=` reduction on **`int`** `n`, then after the loop `n` holds the summation of its value before the loop started and all right-hand sides that the loop supplied. Within the loop, the name `n` that is being reduced to cannot be read or written. It may only be reduced to further with the same operator.

The minimising **`min`**= and maximising **`max`**= reductions have an extended form with extra arguments [red-c], which are supplied on both sides between angle brackets. These extra arguments are saved whenever a new minimum or maximum, respectively, is found.

Reduction Expressions. Every reduction assignment has a corresponding expression form [num-red][bool-red]. For `+=` there is **`sum`**, for `|=` there is **`any`** etc. These still define an iterator, range and filter like a loop, but only define the expression that needs to be reduced.

[1] The type rules for outgoing ranges are those on ingoing edges with the range name changed. The same holds for down and up ranges. Therefore we have not included those rules in the figure.

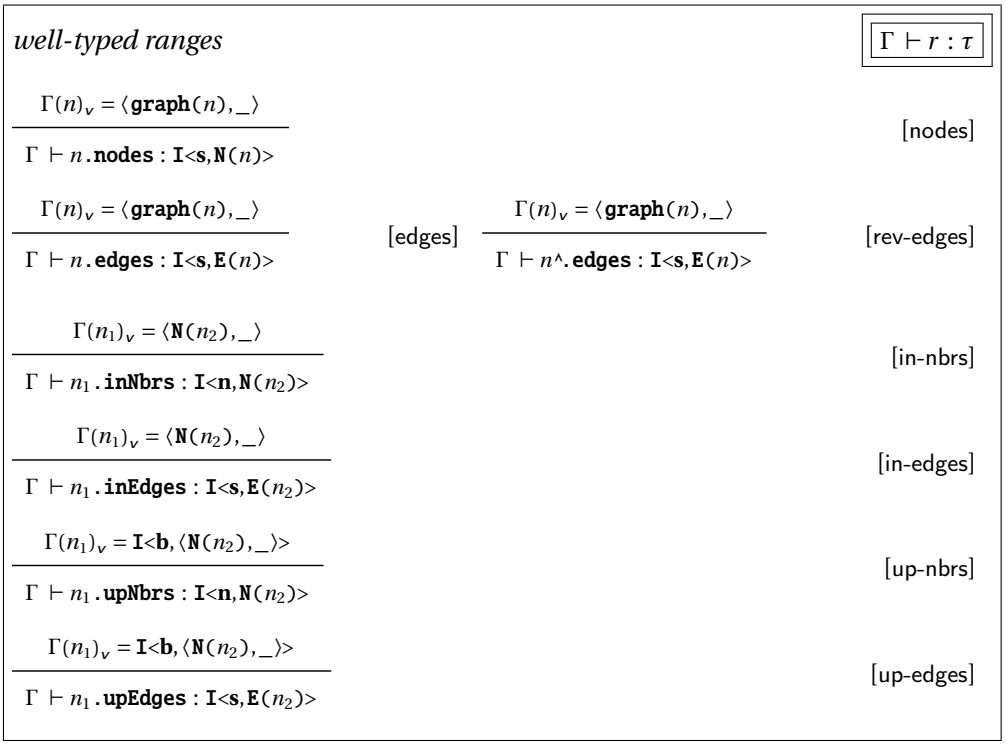


Figure 3.9 – Graph related ranges.

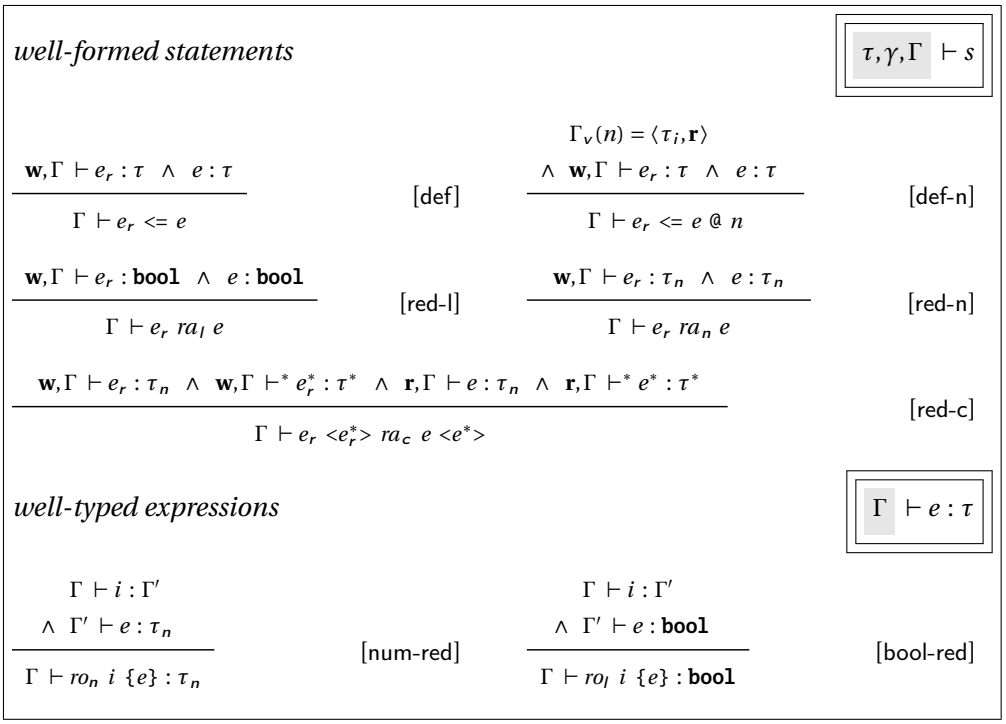


Figure 3.10 – Reductions assignments and expressions.

syntax

t_c	=	**N_S** | **N_S**(n) | **E_S** | **E_S**(n)	sets
	|	**N_Q** | **N_Q**(n) | **E_Q** | **E_Q**(n)	sequences
	|	**N_O** | **N_O**(n) | **E_O** | **E_O**(n)	orders
t_{cc}	=	**collection**<t_c>	collections of collections
r	=	...	
	|	n.**items** | n^.**items**	collection ranges

semantic domains

τ_c	=	**S**<τ_e> | **Q**<τ_e> | **O**<τ_e>	graph collections
τ_{cc}	=	**Q**<τ_c>	collections of collections
ι	=	...	iterator context
	|	**c**	collection access

Figure 3.11 – Syntax of collections.

3.4 Collections

There are three kinds of basic collections and a collection of basic collections in GREEN-MARL. Basic collections are sets, sequences and orders of graph elements. Each of the types is shown in Figure 3.11 along with the syntax for ranging over a collection and the related semantic domains.

Sets, Sequences and Orders. *Sets* have unique elements but no ordering between their elements. *Sequences* do not have unique elements but do have an ordering. *Orders* have both unique elements and an ordering. These elements can only be nodes or edges of a graph, and only those that are all of the same graph. The graph name is used in the type and can be inferred if only one graph is in scope (e.g. [N-S-i] in Figure 3.12). The semantic types describe only the three different kinds of collections and take a graph element type as a type parameter, rather than having six specialised types.

Collections of collections. The collection of collections is a *sequence* of a single kind of basic collection [CC]. The basic collections must always be on the same graph element and related to the same graph.

Ranges. All collections are iterable, through the **items** range. These ranges produce a collection iterator **c**. This information is used in a later stage by a static analysis that's described separate from the type system in Chapter 4.

An ordered collection is also iterable in reverse with the ^.

3.5 Graph properties

Beside collections, there is a graph related mapping type in GREEN-MARL. It is graph-element specific like the collections and is called a graph property. The syntax and related types are in Figure 3.13.

semantic type translation $\boxed{\gamma \vdash t \Rightarrow \tau}$

$\langle \ldots, n, \ldots \rangle \vdash \texttt{N\_S}(n) \Rightarrow \texttt{S<N}(n)\texttt{>}$ [N-S]

$\langle n \rangle \vdash \texttt{N\_S} \Rightarrow \texttt{S<N}(n)\texttt{>}$ [N-S-i]

$\langle \ldots, n, \ldots \rangle \vdash \texttt{E\_S}(n) \Rightarrow \texttt{S<E}(n)\texttt{>}$ [E-S]

$\langle n \rangle \vdash \texttt{E\_S} \Rightarrow \texttt{S<E}(n)\texttt{>}$ [E-S-i]

$\langle \ldots, n, \ldots \rangle \vdash \texttt{N\_Q}(n) \Rightarrow \texttt{Q<N}(n)\texttt{>}$ [N-Q]

$\langle n \rangle \vdash \texttt{N\_Q} \Rightarrow \texttt{Q<N}(n)\texttt{>}$ [N-Q-i]

$\langle \ldots, n, \ldots \rangle \vdash \texttt{E\_Q}(n) \Rightarrow \texttt{Q<E}(n)\texttt{>}$ [E-Q]

$\langle n \rangle \vdash \texttt{E\_Q} \Rightarrow \texttt{Q<E}(n)\texttt{>}$ [E-Q-i]

$\langle \ldots, n, \ldots \rangle \vdash \texttt{N\_O}(n) \Rightarrow \texttt{O<N}(n)\texttt{>}$ [N-O]

$\langle n \rangle \vdash \texttt{N\_O} \Rightarrow \texttt{O<N}(n)\texttt{>}$ [N-O-i]

$\langle \ldots, n, \ldots \rangle \vdash \texttt{E\_O}(n) \Rightarrow \texttt{O<E}(n)\texttt{>}$ [E-O]

$\langle n \rangle \vdash \texttt{E\_O} \Rightarrow \texttt{O<E}(n)\texttt{>}$ [E-O-i]

$$\frac{\gamma \vdash t_c \Rightarrow \tau_c}{\gamma \vdash \texttt{collection<}t_c\texttt{>} \Rightarrow \texttt{Q<}\tau_c\texttt{>}}$$ [CC]

well-typed ranges $\boxed{\Gamma \vdash r : \tau}$

$$\frac{\Gamma_v(n) = \langle \texttt{S<}\tau_e\texttt{>}, \_ \rangle}{\Gamma \vdash n\texttt{.items} : \texttt{I<c,}\tau_e\texttt{>}}$$ [s-items]

$$\frac{\Gamma_v(n) = \langle \texttt{Q<}\tau_e\texttt{>}, \_ \rangle}{\Gamma \vdash n\texttt{.items} : \texttt{I<c,}\tau_e\texttt{>}}$$ [q-items]

$$\frac{\Gamma_v(n) = \langle \texttt{Q<}\tau_e\texttt{>}, \_ \rangle}{\Gamma \vdash n\texttt{\^{}.items} : \texttt{I<c,}\tau_e\texttt{>}}$$ [r-q-items]

$$\frac{\Gamma_v(n) = \langle \texttt{O<}\tau_e\texttt{>}, \_ \rangle}{\Gamma \vdash n\texttt{.items} : \texttt{I<c,}\tau_e\texttt{>}}$$ [o-items]

$$\frac{\Gamma_v(n) = \langle \texttt{O<}\tau_e\texttt{>}, \_ \rangle}{\Gamma \vdash n\texttt{\^{}.items} : \texttt{I<c,}\tau_e\texttt{>}}$$ [r-o-items]

$$\frac{\Gamma_v(n) = \langle \texttt{Q<}\tau_c\texttt{>}, \_ \rangle}{\Gamma \vdash n\texttt{.items} : \texttt{I<c,}\tau_c\texttt{>}}$$ [c-items]

$$\frac{\Gamma_v(n) = \langle \texttt{Q<}\tau_c\texttt{>}, \_ \rangle}{\Gamma \vdash n\texttt{\^{}.items} : \texttt{I<c,}\tau_c\texttt{>}}$$ [r-c-items]

Figure 3.12 – Collection translations and ranges.

syntax

t_g	=	$\texttt{N\_P}\langle t_{pc}\rangle \mid \texttt{N\_P}\langle t_{pc}\rangle(n)$	node properties
	|	$\texttt{E\_P}\langle t_{pc}\rangle \mid \texttt{E\_P}\langle t_{pc}\rangle(n)$	edge properties
t_{pc}	=	$t_p \mid t_c \mid t_{cc}$	property targets
e_r	=	$\ldots$	
	|	$n.n$	property access
	|	$\ldots$	

semantic domains

τ_g	=	$\texttt{P}\langle \tau_e, \tau_{pc}\rangle$	graph properties
τ_{pc}	=	$\tau_p \mid \tau_c \mid \tau_{cc}$	property targets
Γ_g	=	$n \times n \rightarrow_{fin} (\tau \times \alpha)$	graph property environment

judgements

$\vdash \tau \Rightarrow n$ graph reference extraction

Figure 3.13 – Syntax of graph properties.

Properties. Graph properties are a full mapping from nodes or edges to some primitive or collection type. It is a full mapping of all nodes or edges (keys) in a graph . The result of accessing a property with `NIL` is undefined.

A graph property is defined like any other local variable, but gets added to the graph property environment. Graph properties can be overloaded on different graphs, therefore the graph property environment takes the name of the property and of the graph as keys. We use an auxiliary judgement to extract the graph reference out of properties to avoid duplicating the rules too much [p-ex][n-ex][r-ex].

A graph property is referenced using a dot-access syntax. For a node property the node comes before the dot and the graph property comes after the dot. , therefore the graph property environment uses both the name of the property and the graph it belongs to as the way to look up the graph property type in [prop-r] and [prop-n] in Figure 3.14.

Note that graph properties *cannot* be overloaded on graph element, because that would make group assignment on graphs ambiguous.

Group assignments. You can use property assignment syntax to assign to multiple keys concurrently by supplying either a collection of them [gs][gq][go] or the entire graph [gg]. The right-hand side of such a group assignment has access to the 'current' key of the group through a placeholder called `_`.

3.6 Maps

Another mapping type is the more general map between two ordered types. This is a partial mapping as opposed to the full mapping of the graph property types from the last section. Both the syntax and the rules are combined into Figure 3.15.

Maps. Maps define a partial mapping between keys and values. Values need to be types that have an ordering defined and keys need to be types that have equality defined. The API for maps (Section 3.8) offers functions to get the largest and smallest value or key that maps to that value. The types with an ordering are all primitive types t_p and the graph-element types t_e except Booleans, which are excluded in translation rule [m].

3.7 Top-Level Declarations

GREEN-MARL has compilation units consisting of procedure declarations. This section describes the definition and call of procedures. The syntax and semantic domains are Figure 3.16.

Procedure Declarations. Procedures are defined by a name, a list of named *in*-arguments, a list of named *out*-arguments, optionally a return type, and the body statements. In-arguments are normal procedure arguments, and are read-only in GREEN-MARL. Out-arguments can be used both for supplying more arguments and for returning multiple values from a procedure.

In-arguments to a procedure are the only way to introduce graphs in a GREEN-MARL program [arg-t-gr]. Because these graphs' names can be used in the types of

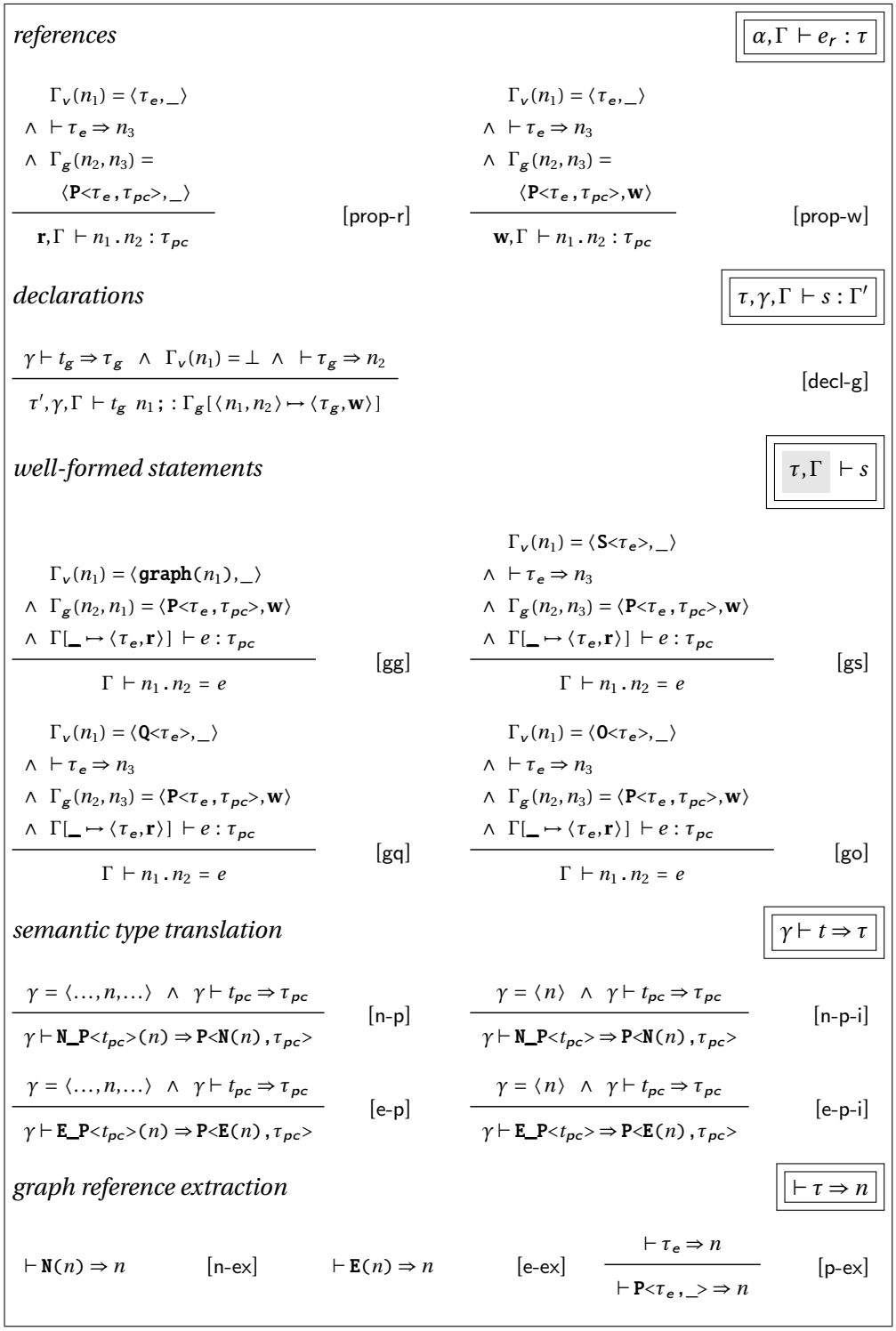


Figure 3.14 – Graph property references and declarations, group assignments and type translations.

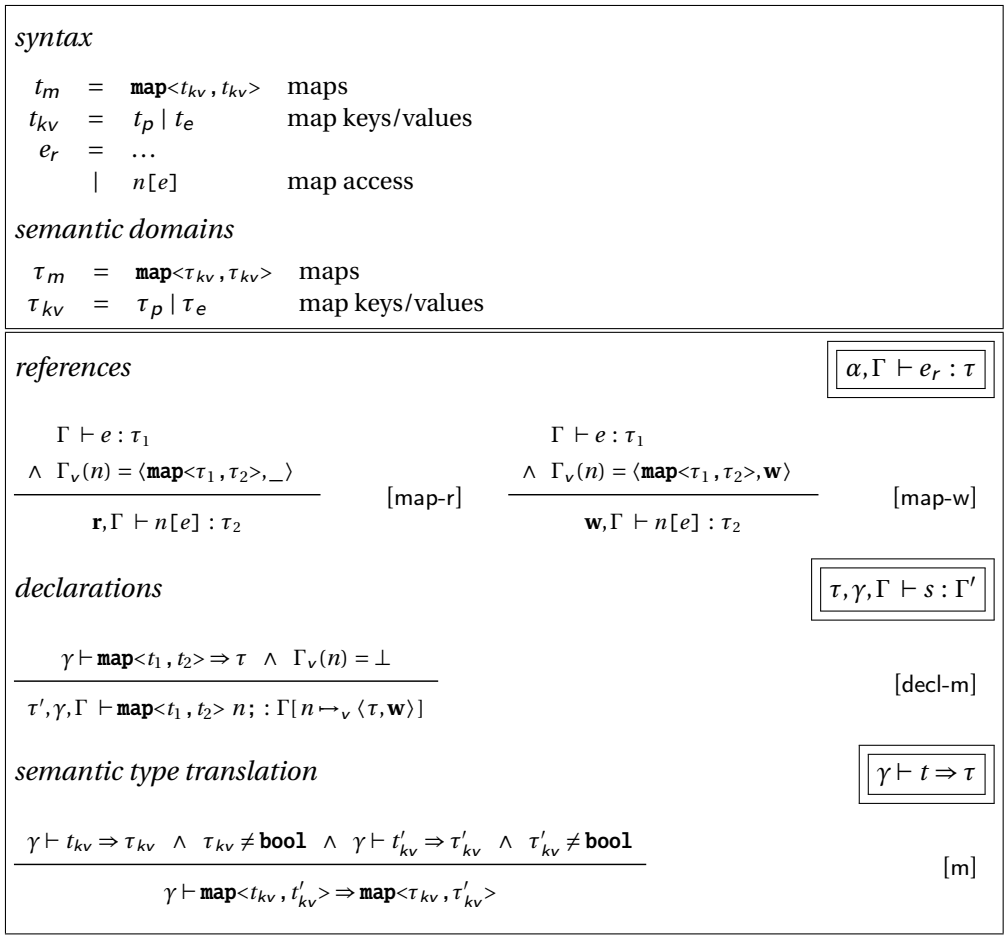


Figure 3.15 – Syntax and rules of maps.

syntax

d	=	**proc** n (**(**f^*; f^***)**: t **{**s^***}**	procedure declarations
	|	**proc** n (**(**f^*; f^***)**{s^***}**	
f	=	n : t	formal arguments
s	=	...	
	|	n(**(**e^*; a^***)**;	procedure calls
	|	**return** e; | **return**;	return statements
	|	...	
e	=	...	
	|	n(**(**e^*; a^***)**)	procedure calls
	|	...	
a	=	e_r	reference arguments
	|	a_i	ignored arguments
a_i	=	**#**	

semantic domains

Γ_p	=	$n \rightarrow_{fin} \sigma_p$	procedure environment
σ	=	$\tau \mid \sigma_p \mid \ldots$	semantic type schemes
σ_p	=	$\forall\ n^*.\ \tau_p$	procedure signatures
τ_p	=	**P**<τ^*,τ^*,τ>	procedure type with variables

semantic judgements

$\alpha, \gamma \vdash f : \tau$	well-typed formal arguments
$\gamma \vdash f : \gamma'$	formal graph arguments
$\alpha, \gamma, \Gamma \vdash f : \Gamma'$	formal argument names
$\Gamma \vdash a : \tau$	well-typed output arguments

Figure 3.16 – Syntax of compilation units, procedures, calls and return statements.

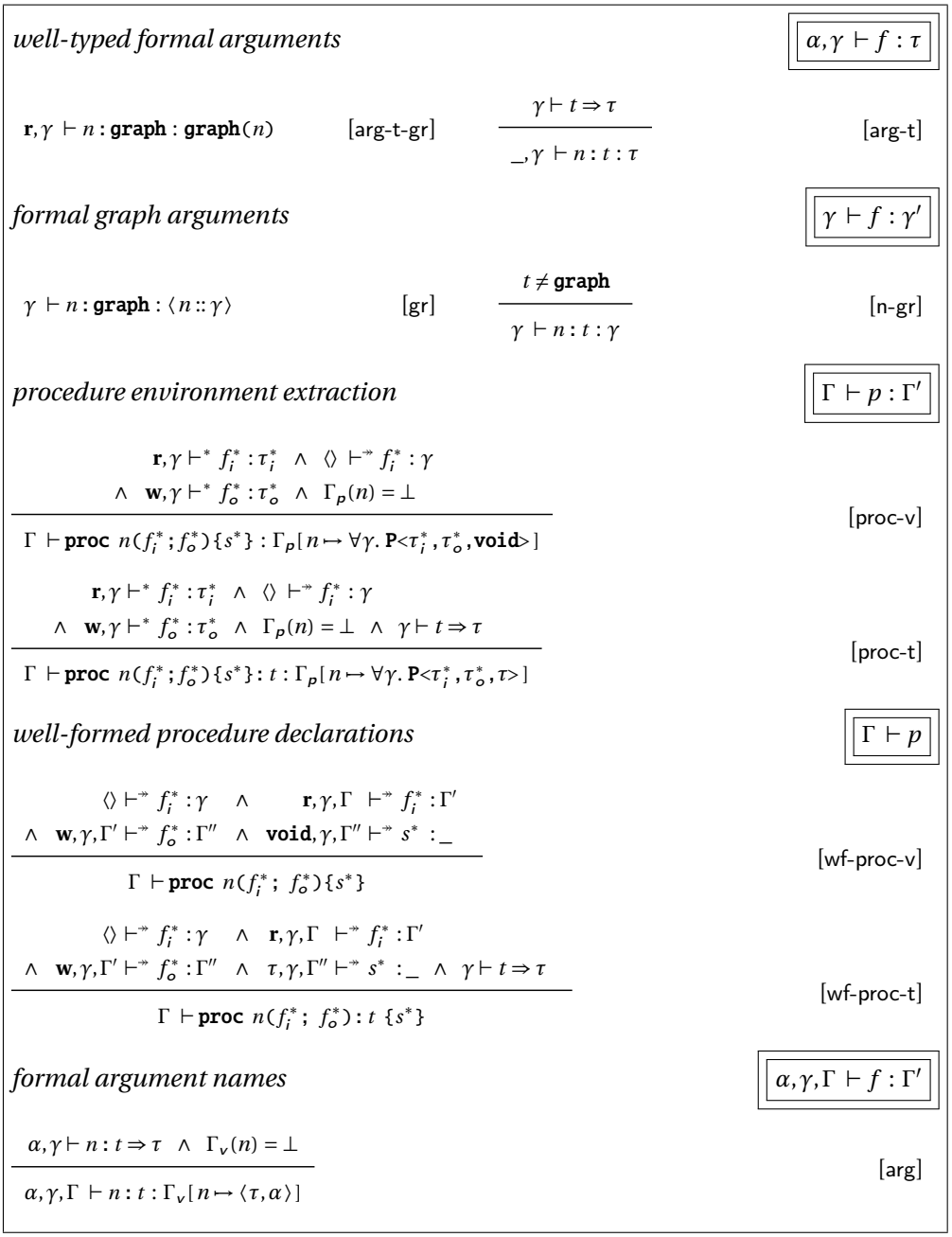


Figure 3.17 – Procedure declaration rules.

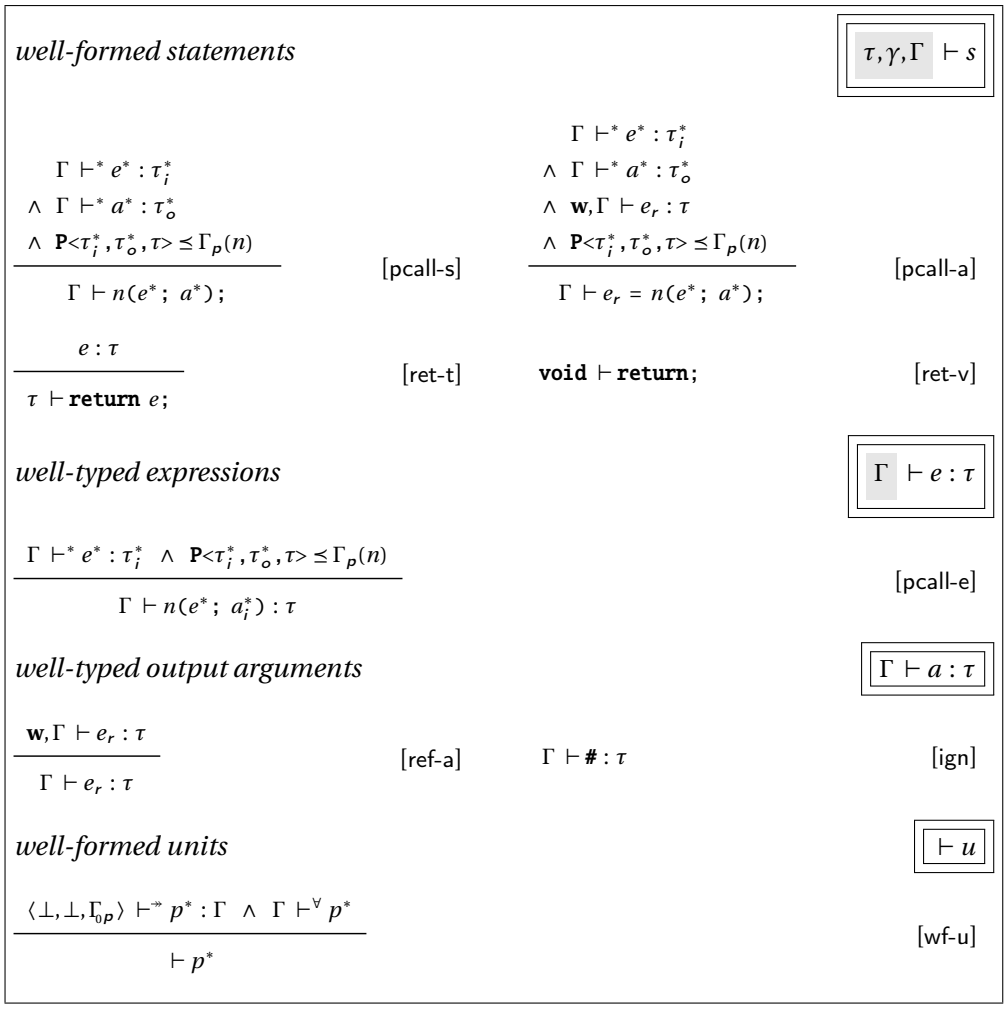


Figure 3.18 – Well-formed statements and units, and well-typed expressions.

other arguments, we need to collect these graph names separately [gr] (Figure 3.17) and use them to check the correctness of the other types. Since procedures can be called, graphs can be arguments, and the graph argument name can be used in the types of other arguments, the type of a procedure has to be polymorphic in the graph name [proc-v]. This is where we first use type schemes, types with a universal quantification.

Procedure Calls. Procedures are called with expressions e^* for the in-arguments and references a^* for the out-arguments. Only these are special references that can also be the ignore symbol **#**. The ignore symbol is given an arbitrary type so it can always be used ([ign] in Figure 3.18).

The types of the arguments are gathered and turned into a type signature for a procedure, which is judged to be an instantiation $\preceq$ of the type scheme that the environment has of the procedure in question [pcall-s][pcall-a].

Note that procedure calls on the expression level can only have ignore symbols a_i^* for out-arguments. GREEN-MARL uses this to guarantee that expressions do not have side-effects.

Return Statements. The return type τ of the procedure is provided in all well-formedness judgements of statements as an optional argument ([wf-proc-v] and [wf-proc-t]). It is implicitly passed down and only used by the return statement [ret-t]. The semantic type `void` is used to describe a procedure with no return type [ret-v].

Compilation units. The top-level compilation unit of GREEN-MARL is based on a file with procedure declarations. The procedure names are collected in an environment Γ_p, on top of the built-in procedures Γ_{0p}. These built-ins are defined in Section 3.8. Note that user-defined procedures are not allowed to be overloaded [wf-u].

The well-formed units rule checks the well-formedness of all procedures p^* using $\vdash^\forall$. In other procedure-related rules, we used checked well-typedness of lists by mapping a judgement with $\vdash^*$. By example, these expand to:

$$\frac{\Gamma \vdash p_1 \ \wedge \ \ldots \ \wedge \ \Gamma \vdash p_n}{\Gamma \vdash^\forall \langle p_1, \ldots, p_n \rangle} \quad \text{[forall]}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \ \wedge \ \ldots \ \wedge \ \Gamma \vdash e_n : \tau_n}{\Gamma \vdash^* \langle e_1, \ldots, e_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle} \quad \text{[map]}$$

3.8 Functions and API

GREEN-MARL has syntax for function calls, but not for function definitions (Figure 3.19). In this section we discuss function calls and the predefined functions and procedures.

Function Calls. Function calls are different from procedure calls in that they are called on a subject expression e, only take in-arguments and always have a result type. Functions cannot be defined by users, instead GREEN-MARL provides some built-in functions in Γ_{0f}. These built-in functions can be polymorphic in graph name or types, and the function call rule [fcall-e] uses the instantiation judgement similarly to the procedure call rules in Section 3.7. We use $\hat{\tau}$ for type variables and $\check{\tau}$ for types with type variables.

Note that Γ_{0f} is a *multi-valued* function. A built-in function can be overloaded, and we consider the lookup of a function non-deterministic. In [fcall-e] the instantiation judgement collapses the non-determinism.

Mathematics. There are a number of top-level math procedures defined: uniformly distributed random numbers, random numbers within a range, base e logarithm and exponentiation, square root, and power; these are all shown with their types in Figure 3.20.

Strings. Strings may be queried for their length, substrings, prefixes, postfixes and possibly case-insensitive equality.

Date and Time. The date and time procedures supply the current time, parsing functionality, time difference, extract of components, and setting the default calendar and parsing format.

syntax

$$
\begin{array}{lcll}
s & = & \ldots & \\
 & | & e.n(e^*); & \text{function calls} \\
e & = & \ldots & \\
 & | & e.n(e^*) & \text{function calls}
\end{array}
$$

semantic domains

$$
\begin{array}{lcll}
\Gamma_f & = & n \rightarrow_{fin} \sigma_f^* & \text{built-in functions} \\
\sigma & = & \ldots \mid \sigma_f & \text{semantic type schemes} \\
\sigma_f & = & \forall\, n^*, \hat{\tau}^*.\, \check{\tau}_f & \text{function signatures} \\
\tau_f & = & \mathbf{F}\langle \tau, \tau^*, \tau \rangle & \text{function type with variables} \\
\hat{\tau} & & & \text{type variables} \\
\check{\tau} & & & \text{types with variables}
\end{array}
$$

well-formed statements $\boxed{\tau, \gamma, \Gamma \vdash s}$

$$\frac{\Gamma \vdash e.n(e^*) : \tau}{\Gamma \vdash e.n(e^*);}$$ [fcall-s]

well-typed expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\begin{array}{l} e : \tau_e \ \wedge \ e^* : \tau^* \\ \wedge \ \mathbf{F}\langle \tau_e, \tau^*, \tau \rangle \preceq \Gamma_{0f}(n) \end{array}}{e.n(e^*) : \tau}$$ [fcall-e]

Figure 3.19 – Syntax of functions calls, and their well-formedness and well-typedness.

Built-in math procedures

Γ_{0p}(uniform) = P<⟨⟩,⟨⟩,**double**>

Γ_{0p}(rand) = P<⟨**long**⟩,⟨⟩,**long**>

Γ_{0p}(log) = P<⟨**double**⟩,⟨⟩,**double**>

Γ_{0p}(exp) = P<⟨**double**⟩,⟨⟩,**double**>

Γ_{0p}(sqrt) = P<⟨**double**⟩,⟨⟩,**double**>

Γ_{0p}(pow) = P<⟨**double**,**double**⟩,⟨⟩,**double**>

Built-in **string** *functions*

Γ_{0f}(length) = F<**string**,⟨⟩,**int**>

Γ_{0f}(contains) = F<**string**,⟨**string**⟩,**bool**>

Γ_{0f}(beginsWith) = F<**string**,⟨**string**⟩,**bool**>

Γ_{0f}(endsWith) = F<**string**,⟨**string**⟩,**bool**>

Γ_{0f}(equals) = F<**string**,⟨**string**,**bool**⟩,**bool**>

Built-in **date** *procedures*

Γ_{0p}(currentTime) = P<⟨⟩,⟨⟩,**date**>

Γ_{0p}(parseTime) = P<⟨**string**,**string**,**string**⟩,⟨⟩,**date**>

Γ_{0p}(parseTimeWithFormat) = P<⟨**string**,**string**⟩,⟨⟩,**date**>

Γ_{0p}(diffTime) = P<⟨**date**,**date**,**string**⟩,⟨⟩,**double**>

Γ_{0p}(extractTime) = P<⟨**date**,**string**⟩,⟨⟩,**int**>

Γ_{0p}(setDefaultCalendarSystem) = P<⟨**string**⟩,⟨⟩,**void**>

Γ_{0p}(setDefaultTimeLiteralFormat) = P<⟨**string**⟩,⟨⟩,**void**>

Figure 3.20 – Built-ins for mathematics, **date** and **string**.

Graphs, nodes and edges. The functions for graphs and nodes can count nodes, edges and neighbours, and pick random nodes and neighbours. The `pickRandom` showcases the need to have the semantic graph type refer to the name of the graph, because otherwise the node that is returned cannot be typed with the right graph.

The functions for edges provide the start and end of an edge. The functions for neighbour iterators provide the edge in between the origin node of the iteration and the current neighbour. These neighbour function show the use of remembering the iterator is a neighbour iterator, as well as a reason why we even introduced iterators as a *type* instead of extra context information.

Collections and Maps. The Collections and Maps API showcase overloading of functions and polymorphism in types. The functions on the collections are polymorphic in the type of element in the collection. The map API is polymorphic in the types of both the keys and the values.

graphs

Γ_{0f}(numNodes) $= \forall n.\ \mathbf{F}\langle\mathbf{graph}(n), \langle\rangle, \mathbf{int}\rangle$

Γ_{0f}(numEdges) $= \forall n.\ \mathbf{F}\langle\mathbf{graph}(n), \langle\rangle, \mathbf{int}\rangle$

Γ_{0f}(pickRandom) $= \forall n.\ \mathbf{F}\langle\mathbf{graph}(n), \langle\rangle, \mathbf{N}(n)\rangle$

nodes

Γ_{0f}(pickRandomNbr) $= \forall n.\ \mathbf{F}\langle\mathbf{N}(n), \langle\rangle, \mathbf{N}(n)\rangle$

Γ_{0f}(numInNbrs) $= \forall n.\ \mathbf{F}\langle\mathbf{N}(n), \langle\rangle, \mathbf{int}\rangle$

Γ_{0f}(numOutNbrs) $= \forall n.\ \mathbf{F}\langle\mathbf{N}(n), \langle\rangle, \mathbf{int}\rangle$

Γ_{0f}(hasEdgeFrom) $= \forall n.\ \mathbf{F}\langle\mathbf{N}(n), \langle\mathbf{N}(n)\rangle, \mathbf{bool}\rangle$

Γ_{0f}(hasEdgeTo) $= \forall n.\ \mathbf{F}\langle\mathbf{N}(n), \langle\mathbf{N}(n)\rangle, \mathbf{bool}\rangle$

edges

Γ_{0f}(fromNode) $= \forall n.\ \mathbf{F}\langle\mathbf{E}(n), \langle\rangle, \mathbf{N}(n)\rangle$

Γ_{0f}(toNode) $= \forall n.\ \mathbf{F}\langle\mathbf{E}(n), \langle\rangle, \mathbf{N}(n)\rangle$

neighbour iterators

Γ_{0f}(fromEdge) $= \forall n.\ \mathbf{F}\langle\mathbf{I}\langle\mathbf{n}, \mathbf{N}(n)\rangle, \langle\rangle, \mathbf{E}(n)\rangle$

Γ_{0f}(toEdge) $= \forall n.\ \mathbf{F}\langle\mathbf{I}\langle\mathbf{n}, \mathbf{N}(n)\rangle, \langle\rangle, \mathbf{E}(n)\rangle$

Figure 3.21 – Graph-related API.

Individual functions such as `size`, `has` and the entire sequence API are overloaded for different types. In the case of the sequence API it is shared by sequences, orders and collections of collections. The set function `add` is an example of an overloaded function for the same type, where one adds a single element to a set, and the other adds a whole other set and is an alias of `addAll`.

set functions

Γ_{0f}(size) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{S}\langle \hat{t}\rangle, \langle\rangle, \mathbf{int}\rangle$

Γ_{0f}(has) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{S}\langle \hat{t}\rangle, \langle\rangle, \mathbf{bool}\rangle$

Γ_{0f}(isSubsetOf) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{S}\langle \hat{t}\rangle, \langle \mathbf{S}\langle \hat{t}\rangle\rangle, \mathbf{bool}\rangle$

Γ_{0f}(add) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{S}\langle \hat{t}\rangle, \langle \hat{t}\rangle, \mathbf{void}\rangle$

Γ_{0f}(add) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{S}\langle \hat{t}\rangle, \langle \mathbf{S}\langle \hat{t}\rangle\rangle, \mathbf{void}\rangle$

Γ_{0f}(remove) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{S}\langle \hat{t}\rangle, \langle \hat{t}\rangle, \mathbf{void}\rangle$

Γ_{0f}(remove) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{S}\langle \hat{t}\rangle, \langle \mathbf{S}\langle \hat{t}\rangle\rangle, \mathbf{void}\rangle$

Γ_{0f}(addAll) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{S}\langle \hat{t}\rangle, \langle \mathbf{S}\langle \hat{t}\rangle\rangle, \mathbf{void}\rangle$

Γ_{0f}(removeAll) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{S}\langle \hat{t}\rangle, \langle \mathbf{S}\langle \hat{t}\rangle\rangle, \mathbf{void}\rangle$

Γ_{0f}(retainOnly) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{S}\langle \hat{t}\rangle, \langle \mathbf{S}\langle \hat{t}\rangle\rangle, \mathbf{void}\rangle$

Γ_{0f}(clear) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{S}\langle \hat{t}\rangle, \langle\rangle, \mathbf{void}\rangle$

sequence functions

Γ_{0f}(size) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{Q}\langle \hat{t}\rangle, \langle\rangle, \mathbf{int}\rangle$

Γ_{0f}(has) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{Q}\langle \hat{t}\rangle, \langle\rangle, \mathbf{bool}\rangle$

Γ_{0f}(front) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{Q}\langle \hat{t}\rangle, \langle\rangle, \hat{t}\rangle$

Γ_{0f}(back) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{Q}\langle \hat{t}\rangle, \langle\rangle, \hat{t}\rangle$

Γ_{0f}(pop) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{Q}\langle \hat{t}\rangle, \langle\rangle, \hat{t}\rangle$

Γ_{0f}(popFront) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{Q}\langle \hat{t}\rangle, \langle\rangle, \hat{t}\rangle$

Γ_{0f}(popBack) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{Q}\langle \hat{t}\rangle, \langle\rangle, \hat{t}\rangle$

Γ_{0f}(push) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{Q}\langle \hat{t}\rangle, \langle \hat{t}\rangle, \mathbf{void}\rangle$

Γ_{0f}(pushFront) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{Q}\langle \hat{t}\rangle, \langle \hat{t}\rangle, \mathbf{void}\rangle$

Γ_{0f}(pushFront) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{Q}\langle \hat{t}\rangle, \langle \mathbf{Q}\langle \hat{t}\rangle\rangle, \mathbf{void}\rangle$

Γ_{0f}(pushBack) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{Q}\langle \hat{t}\rangle, \langle \hat{t}\rangle, \mathbf{void}\rangle$

Γ_{0f}(pushBack) $= \forall \hat{t}.\ \mathbf{F}\langle \mathbf{Q}\langle \hat{t}\rangle, \langle \mathbf{Q}\langle \hat{t}\rangle\rangle, \mathbf{void}\rangle$

map functions

Γ_{0f}(size) $= \forall \hat{t}_1, \hat{t}_2.\ \mathbf{F}\langle \mathbf{map}\langle \hat{t}_1, \hat{t}_2\rangle, \langle\rangle, \mathbf{int}\rangle$

Γ_{0f}(hasKey) $= \forall \hat{t}_1, \hat{t}_2.\ \mathbf{F}\langle \mathbf{map}\langle \hat{t}_1, \hat{t}_2\rangle, \langle \hat{t}_1\rangle, \mathbf{bool}\rangle$

Γ_{0f}(hasMinValue) $= \forall \hat{t}_1, \hat{t}_2.\ \mathbf{F}\langle \mathbf{map}\langle \hat{t}_1, \hat{t}_2\rangle, \langle \hat{t}_1\rangle, \mathbf{bool}\rangle$

Γ_{0f}(hasMaxValue) $= \forall \hat{t}_1, \hat{t}_2.\ \mathbf{F}\langle \mathbf{map}\langle \hat{t}_1, \hat{t}_2\rangle, \langle \hat{t}_1\rangle, \mathbf{bool}\rangle$

Γ_{0f}(getMaxKey) $= \forall \hat{t}_1, \hat{t}_2.\ \mathbf{F}\langle \mathbf{map}\langle \hat{t}_1, \hat{t}_2\rangle, \langle\rangle, \hat{t}_1\rangle$

Γ_{0f}(getMinKey) $= \forall \hat{t}_1, \hat{t}_2.\ \mathbf{F}\langle \mathbf{map}\langle \hat{t}_1, \hat{t}_2\rangle, \langle\rangle, \hat{t}_1\rangle$

Γ_{0f}(getMaxValue) $= \forall \hat{t}_1, \hat{t}_2.\ \mathbf{F}\langle \mathbf{map}\langle \hat{t}_1, \hat{t}_2\rangle, \langle\rangle, \hat{t}_2\rangle$

Γ_{0f}(getMinValue) $= \forall \hat{t}_1, \hat{t}_2.\ \mathbf{F}\langle \mathbf{map}\langle \hat{t}_1, \hat{t}_2\rangle, \langle\rangle, \hat{t}_2\rangle$

Figure 3.22 – Collection and Map APIs.

Chapter 4

Read-Write Analysis

In the last chapter we introduced the formal specification of GREEN-MARL's type system. In this chapter we extend these with judgements that formalise the dependence analysis and use this information to check the invariants of the reduce and the defer assignments.

General concept. The read-write analysis is a tree-based, symbolic analysis. The tree-based aspect gives us rules that define the read-write information for every statement. Block statements scope local names in sub-statements, therefore those are eliminated from the read-write information in blocks. In general only the outside observable effects of a statement are in the read-write information. This allows for a bottom-up approach that abstracts as it goes up.

The symbolic aspect of the analysis revolves around properties and traversals. Properties are considered special cases because they represent a large amount of individually accessible information under one name. The name of the accessor of a property is kept in the read-write information for as long as possible. When it goes out of scope, an approximate access pattern is saved instead. The accessor is kept symbolically, because properties are accessed by nodes and edges, which do not have literals. Most nodes and edges in a GREEN-MARL program come from a traversals that ranges over a part of the graph. We can approximate the behaviour of a traversal over a range well enough in some cases, that we can apply optimisations such as loop fusion.

The semantic domains and judgements of this chapter are in Figure 4.1. The central judgements are:

- *Expression analysis:* $\vdash_{\mathsf{rw}} e : \varrho$
  We judge expressions e to have a read-write set ϱ. This is a set of 3-tuples of read-write information ri which associates a name n with a read-write mode *mode* and a property access pattern *patt*.

  The access mode includes the standard options **`read`** and **`write`**, and the more domain specific **`defer`**(n) and **`reduce`**(n, ra) which includes information about the bound and the reduction operator.

  The access pattern includes $\not{N}_A$ for normal variables, a name n for access to a single graph element, **`unique`** for access to a unique set of graph elements, **`random`**

for access unpredictable sample of graph elements, including the possibility of accessing the same element multiple times.

- *Statement analysis:* $\Gamma \vdash_{\mathsf{rw}} s : \varrho$
  We judge a statement s to have a read-write set ϱ under environment Γ. Most rules in this chapter are of this judgement.
- *Well-formed read-write sets:* $\vdash_{seq} \varrho, \quad \vdash_{par} \varrho$
  To determine that reduction and deferred assignments are not applied in combination with, for example, normal **`write`** accesses, we employ a well-formedness judgement. This judgement is applied at sequential and parallel traversals.
- *Read-write set transformation:* $n, patt \vdash_{seq} \varrho \Rightarrow \varrho', \quad n, patt \vdash_{par} \varrho \Rightarrow \varrho'$
  At the same traversals where we check the well-formedness of the read-write set, we transform a reduction or deferred assignment to a normal **`write`** if the assignments is bound by the traversal. This transformation is necessary to abstract over the observable effects of the traversal. At this point we also record the property access pattern of the traversal's iterator.

semantic domains

$mode$	$=$	**`read`** | **`write`**	read and write modes
	|	**`defer`**(n)	defer mode
	|	**`reduce`**(n, ra)	reduce mode
$patt$	$=$	N/A	not a property access
	|	n	single point access
	|	**`unique`**	access each point zero or one times
	|	**`random`**	access in an unpredictable way
ra	$=$	$ra_n \mid ra_l$	reduction operators
	|	**`arg`**(ra_c)	min/max argument
o	$=$	$o_n \mid o_c \mid o_e \mid o_l$	binary operators
ϱ	$=$	ri^*	read-write set
ri	$=$	$n \times mode \times patt$	read-write information

semantic judgements

$mode \vdash_{\mathsf{rw}} e_r : \varrho$	reference analysis
$\vdash_{\mathsf{rw}} e : \varrho$	expression analysis
$\vdash_{\mathsf{rw}} r : \varrho$	range analysis
$\Gamma \vdash_{\mathsf{rw}} s : \varrho$	statement analysis
$\Gamma \vdash_{ap} r \Rightarrow patt$	access patterns of ranges
$\vdash_{seq} \varrho$	well-formed sets of sequential traversals
$\vdash_{par} \varrho$	well-formed sets of parallel traversals
$n, patt \vdash_{seq} \varrho \Rightarrow \varrho'$	set transformation for sequential traversals
$n, patt \vdash_{par} \varrho \Rightarrow \varrho'$	set transformation for parallel traversals
$\Gamma \vdash_{mr} ri \Rightarrow ri'$	transformation for properties accessed through local variables

Figure 4.1 – Overview of the semantic domains and judgements.

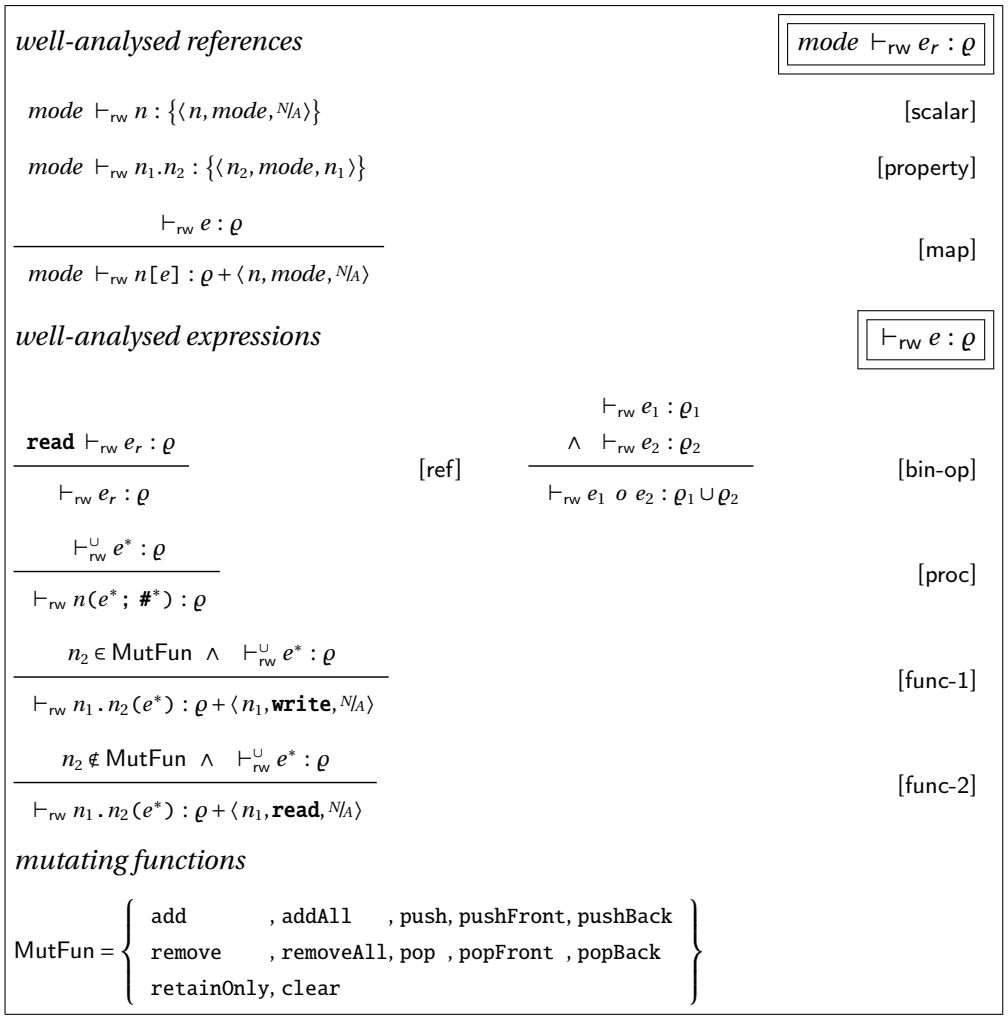


Figure 4.2 – Read-write rules for references, expressions and functions.

4.1 Expressions

We start with the read-write rules for expressions. References are of particular interest, as well as functions.

References and simple expressions. Most expressions have no side-effects on variables. The [ref] rule handles references as **read** (Figure 4.2). The reference rules [scalar], [property] and [map] are configurable by *mode*, this way we can reuse the rules for left-hand sides of assignments. We use the $+$ operator to denote the insertion of a single element into a set.

Simple expressions like a binary operator [bin-op] combine the two operands' read-write sets with a union. The rule for procedures [proc] does the same for a list of expressions e^* by using an abbreviation of judgements denoted with a union symbol. The unabbreviated version of the rule is:

$$\frac{\vdash_{\text{rw}} e_1 : \varrho_1 \;\wedge\; \ldots \;\wedge\; \vdash_{\text{rw}} e_n : \varrho_n}{\vdash_{\text{rw}} n(\langle e_1, \ldots, e_n \rangle;\, \#^*) : \varrho_1 \cup \ldots \cup \varrho_n} \quad \text{[proc-expanded]}$$

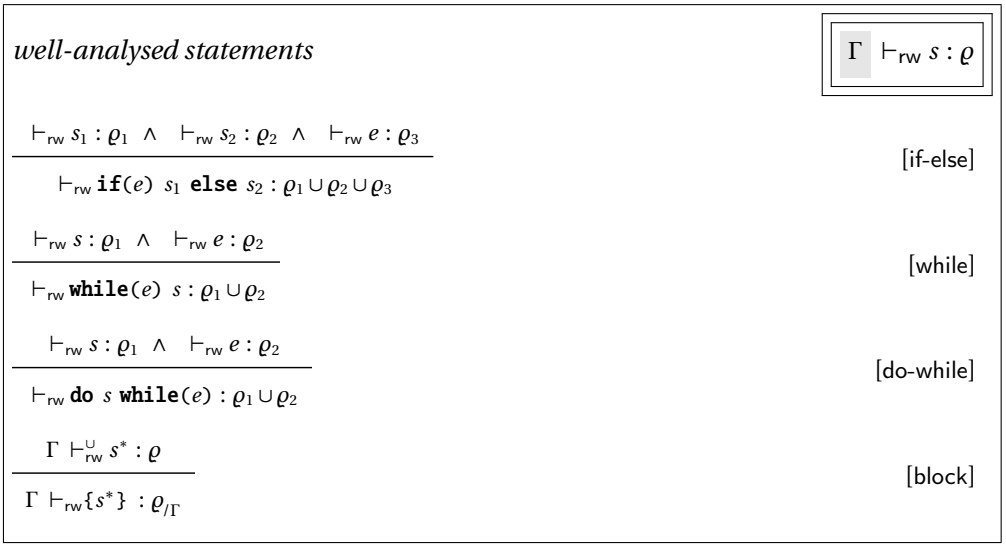


Figure 4.3 – Read-write rules for conditional statements and blocks.

We elide further rules where only unions of sub-expression sets are taken.

Functions. Built-in functions on collections can mutate a collection. For simplicity, we consider this a `write`. The original implementation in the GREEN-MARL compiler treats collection mutation as a special case for better compiler warnings and to inform optimisations. Although function calls are expressions, the ones with a mutation effect have a return type `void`, which makes them only suitable as a statement.

4.2 Statements

Most of our read-write rules relate to statements. They mostly centre around loops and parallel contexts. We start the section with simple conditional statements and the block statement.

Conditionals. The conditional statements have simple rules [if-else], [while] and [do-while] in Figure 4.3. These are comparable to the previous expression statements in that they only union the read-write sets of sub-expressions and sub-statements. The original implementation in the GREEN-MARL compiler tracks conditional read-write information as an extra piece of information, making *ri* a 4-tuple.

Blocks. Block statements drop names from the read-write set which are defined within that block. When these names are used to access into a property, we cannot be sure that a repetition of block accesses the property in the same place, so we use `random` as a conservative value. This transformation of the read-write set is written as a restriction to the environment: $\varrho_{/\Gamma}$. More formally, the restriction is defined as:

$$\varrho_{/\Gamma} = \left\{ ri \;\middle|\; \langle n, mode, patt \rangle \in \varrho \wedge \Gamma(n) \neq \bot \wedge \Gamma \vdash_{mr} \langle n, mode, patt \rangle \Rightarrow ri \right\}$$

where the *mr* judgement is:

$$\frac{\Gamma(n_2) = \bot}{\Gamma \vdash_{mr} \langle n_1, mode, n_2 \rangle \Rightarrow \langle n_1, mode, \texttt{random} \rangle} \quad \text{[restr-trans-1]}$$

$$\frac{\Gamma(n_2) \neq \bot}{\Gamma \vdash_{mr} \langle n_1, mode, n_2 \rangle \Rightarrow \langle n_1, mode, n_2 \rangle} \quad \text{[restr-trans-2]}$$

$$\frac{patt \in \{N\!/\!A, \texttt{unique}, \texttt{random}\}}{\Gamma \vdash_{mr} \langle n_1, mode, patt \rangle \Rightarrow \langle n_1, mode, patt \rangle} \quad \text{[restr-trans-3]}$$

Assignments. Write information originates from assignments. Rule [assign] in Figure 4.4 handles for the normal assignment. This is the second place where we use the judgement for references.

Deferred and reduction assignments are covered by rules [defer] and [reduce] respectively. They depend on an unformalised desugaring step that infers bounds of the assignments, given after the `@`. This inference takes the outermost parallel loop where the left-hand side is still defined (the property if it is a property assignment). When the assignment is not in a parallel context at all, the deferred assignment is bound by the outermost sequential traversal and the reduction assignment is turned into a normal assignment.

The augmented minimum and maximum reduce assignments [arg-min/max] are very similar to the normal reduce assignment rule, except that the extra arguments are annotated as such.

Loops. Iterators from loops can be used in two place: as the accessor of a property, or as bounds for reduction and deferred assignments.

The rules [for] and [foreach] in Figure 4.5 are very similar. Both 1) take the read-write set for the range, 2) get the access pattern of the range, 3) use this range to transform the read-write set of the body statement, 4) union the set of the range

well-analysed statements $\quad \boxed{\Gamma \vdash_{\text{rw}} s : \varrho}$

$$\frac{\vdash_{\text{rw}} e : \varrho_1 \;\wedge\; \texttt{write} \vdash_{\text{rw}} e_r : \varrho_2}{\vdash_{\text{rw}} e_r = e : \varrho_1 \cup \varrho_2} \quad \text{[assign]}$$

$$\frac{\texttt{defer}(n) \vdash_{\text{rw}} e_r : \varrho_1 \;\wedge\; \vdash_{\text{rw}} e : \varrho_2}{\vdash_{\text{rw}} e_r \mathrel{<=} e \;\texttt{@}\; n : \varrho_1 \cup \varrho_2} \quad \text{[defer]}$$

$$\frac{\texttt{reduce}(n, ra) \vdash_{\text{rw}} e_r : \varrho_1 \;\wedge\; \vdash_{\text{rw}} e : \varrho_2}{\vdash_{\text{rw}} e_r \;\; ra \;\; e \;\texttt{@}\; n : \varrho_1 \cup \varrho_2} \quad \text{[reduce]}$$

$$\frac{\begin{array}{l}\texttt{reduce}(n, ra_c) \vdash_{\text{rw}} e_r : \varrho_1 \;\wedge\; \vdash_{\text{rw}} e \;\; : \varrho_3 \\ \wedge\; \texttt{reduce}(n, \texttt{arg}(ra_c)) \vdash^{\cup}_{\text{rw}} e^*_r : \varrho_2 \;\wedge\; \vdash^{\cup}_{\text{rw}} e^* : \varrho_4\end{array}}{\vdash_{\text{rw}} e_r \;\texttt{<}e^*_r\texttt{>}\; ra_c \;\; e \;\texttt{<}e^*\texttt{>}\; \texttt{@}\; n : \varrho_1 \cup \varrho_2 \cup \varrho_3 \cup \varrho_4} \quad \text{[arg-min/max]}$$

Figure 4.4 – Read-write rules for assignments.

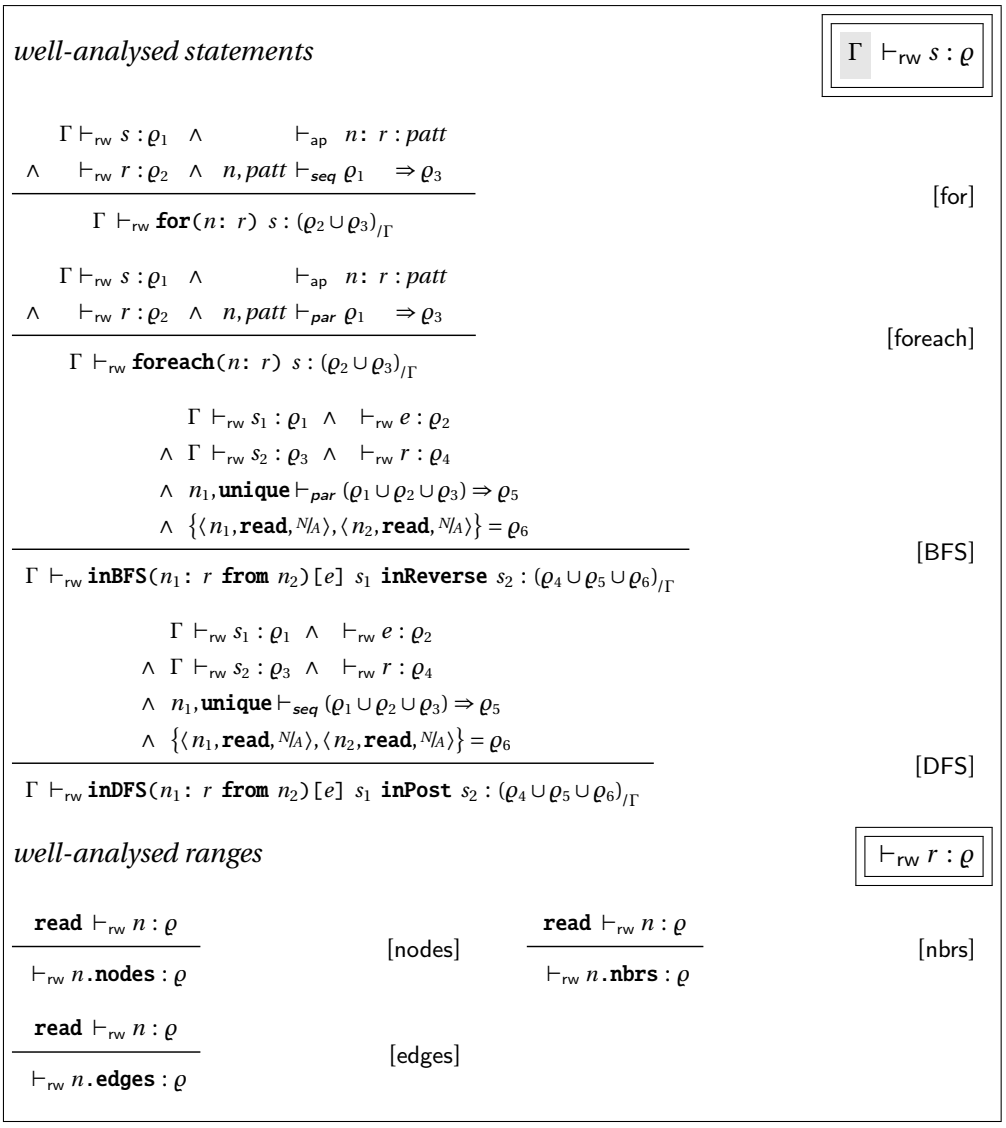


Figure 4.5 – Read-write rules for loops, searches and ranges.

and the transformed set, 5) and finally restrict these two to the environment. The difference between the two is that [for] uses the sequential rule for transformation of the read-write set, whereas [foreach] uses the parallel transformation rule.

The rules for these loops expect a desugared version where the filter expression has been turned into an `if`-statement that wraps the body of the loop.

Searches. The breadth and depth-first search have very similar rules to the loops. These searches have two body statements, a navigator and a range with source node. The union of the bodies' and navigator's sets are transformed. As the actual range of a search is always the same (the graph nodes), we directly use the access pattern of the search, which is **unique**. The depth-first search uses the transformation for sequential traversals, the breadth-first search uses the one for parallel traversals. Again the

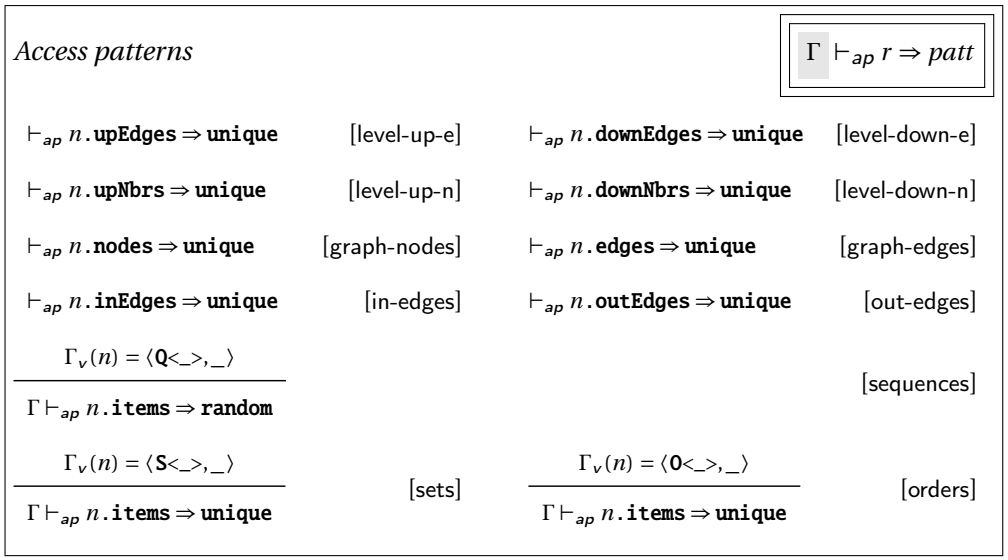


Figure 4.6 – Access patterns by range.

union of the transformed set and the local names from the iterator definition is taken and restricted to the environment.

Access patterns. The access patterns of most ranges is `unique` (Figure 4.6). In fact the only non-unique access pattern is that of [sequences].

4.3 Transformations

The transformations of the read-write analysis that we use for the loops and searches consists of three parts: well-formedness, transformation of the *mode* and transformation of the *patt*. The first is defined in Figure 4.7, the second and third are defined in Figure 4.8 along with the combining rule.

Well-formedness. A read-write set is well-formed when a number of invariants apply. While a variable has a deferred write on it, it should not be written to directly [write-defer]. This is checked on all traversals. On parallel traversals [all-errors], the following other invariants are checked:

1. Reduction assignment target are not written to.
2. Reduction assignment target are not read.
3. Reduction assignment target are not written to in deferred manner.
4. Reduction assignment target are not reduced to with other operators.

Transformation of mode. A deferred and reduction assignment are bound by traversals. Within those traversals they are special assignments, but when we consider their effect from the outside, they simply write to their target. This is what [tr-defer] and [tr-reduce] denote. When the assignment is scoped by the current traversal n_1, we turn the *mode* into `write`.

To keep the rules simple, we do not define the transformation for all read-write information. Instead we use another judgement modifier, that maps a transformation over a set, and preserves the values in the set that the transformation is undefined for:

$$\frac{(n \vdash ri_1 \Rightarrow ri'_1 \textbf{ else } ri'_1 = ri_1) \ \wedge \ \ldots \ \wedge \ (n \vdash ri_n \Rightarrow ri'_n \textbf{ else } ri'_n = ri_n)}{n \vdash^{\circledast} \langle ri_1, \ldots, ri_n \rangle \Rightarrow \langle ri'_1, \ldots, ri'_n \rangle} \quad \text{[map-try]}$$

Transformation of access pattern. The access pattern of an iterator is derived from the range. When we find read-write information on a property that is accessed through the iterator of the traversals that we are at, we simply replace it by its access pattern ([name] in Figure 4.8). The `random` access pattern is simply preserved, so we do not really need [random], but we defined for clarity. Finally a `unique` pattern turns `random` when repeated, as it is possible that a property will be accessed through the same graph element repeatedly.

4.4 Integration with type rules

We have presented the read-write analysis as a separate analysis from the type system of GREEN-MARL. The analysis is orthogonal to the type system, except that it uses the same environment. The read-write analysis does not adapt the environment, it does not influence the type system, therefore we argue that the integration with the type rules is trivial. We give three examples of type rules on the left and fused type rules and read-write rules on the right below:

$$\frac{\tau', \Gamma \vdash s}{\tau', \gamma, \Gamma \vdash s : \Gamma} \quad \text{[non-decl]} \qquad \frac{\tau', \Gamma \vdash_{\mathit{fused}} s : \varrho}{\tau', \gamma, \Gamma \vdash_{\mathit{fused}} s : \Gamma, \varrho} \quad \text{[non-decl-fused]}$$

$$\frac{\Gamma \vdash^{\twoheadrightarrow} s^* : \_}{\Gamma \vdash \{s^*\}} \quad \text{[block]} \qquad \frac{\Gamma \vdash^{\twoheadrightarrow,\cup}_{\mathit{fused}} s^* : \_, \varrho}{\Gamma \vdash_{\mathit{fused}} \{s^*\} : \varrho} \quad \text{[block-fused]}$$

$$\frac{\mathbf{w}, \Gamma \vdash e_r : \tau \ \wedge \ \Gamma \vdash e : \tau}{\Gamma \vdash e_r = e;} \quad \text{[assign]} \qquad \frac{\mathbf{w}, \Gamma \vdash_{\mathit{fused}} e_r : \tau, \varrho_1 \ \wedge \ \Gamma \vdash_{\mathit{fused}} e : \tau, \varrho_2}{\Gamma \vdash_{\mathit{fused}} e_r = e; : \varrho_1 \cup \varrho_2} \quad \text{[assign-fused]}$$

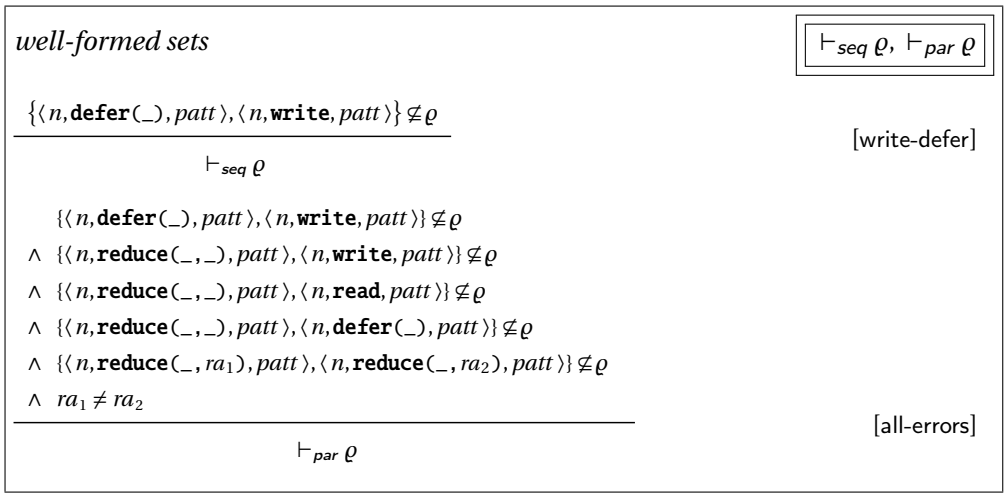


Figure 4.7 – Language invariant checks around defer and reduce.

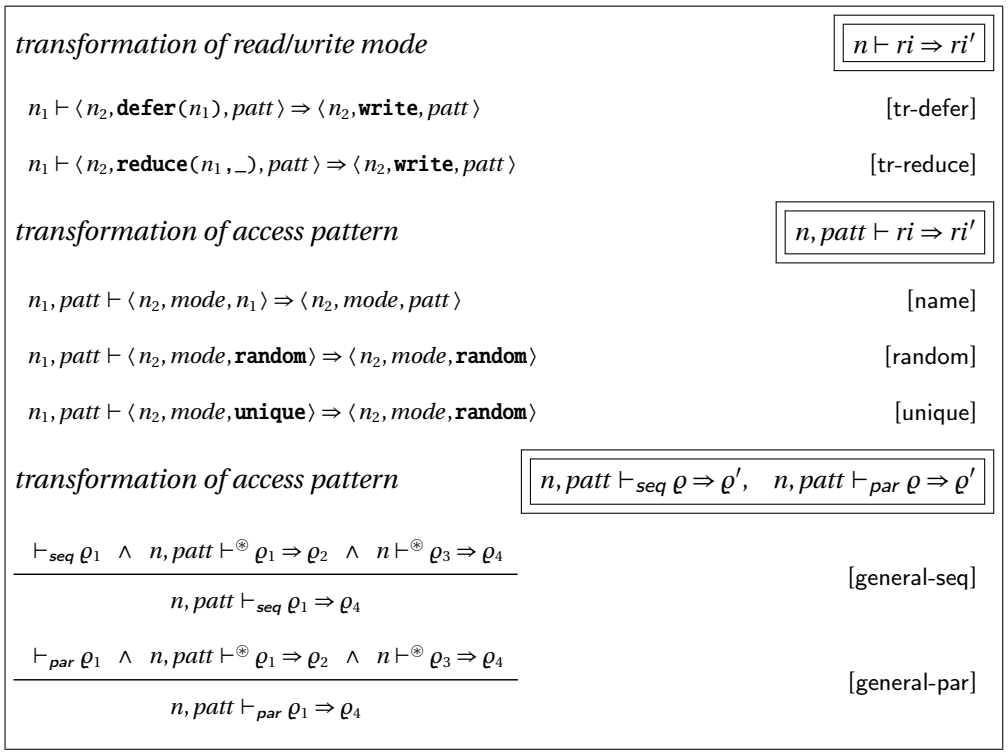


Figure 4.8 – Read-write set transformation.

Chapter 5

Implementation

The original GREEN-MARL compiler `gm_comp` is written in C++. It targets a C++ run-time and a JAVA run-time, and applies a large number of optimisations. We implemented our own compiler `gm_spoofax` that targets the pre-existing JAVA run-time and applies only a subset of `gm_comp`'s optimisations.

We used the Spoofax language workbench [43] to implement `gm_spoofax`. It served the dual purpose of helping us gain a better understanding GREEN-MARL, and giving us an executable specification of GREEN-MARL. This executable specification enabled us to explore how to declaratively specify the static semantics, and see if Spoofax's meta-languages were able to fully and easily capture GREEN-MARL. In this chapter, we report on the challenges we have found during the implementation of the static semantics and compare the implementation with the formal specification.

5.1 Compiler overview

The `gm_spoofax` compiler performs the steps illustrated in Figure 5.1. The boxed steps are this discussed in this chapter. The list of steps is:

1. The compiler starts with GREEN-MARL source code. It *parses* this code into an Abstract Syntax Tree (AST). The parser is derived from a grammar specification in SDF3 [39], Spoofax's syntax definition formalism.
2. This AST is then simplified, *desugared*, with rewrite rules in the STRATEGO transformation language [11]. This is Spoofax's language for term rewrite systems.
3. The desugared AST is analysed. In particular we do a *type analysis*, which is derived from *name binding rules* in NABL [29], *type rules* in TS [38], and *custom extensions* defined in STRATEGO. These rules are applied by an incremental task engine [42], which annotates the tree with results. Notably the AST itself stays the same, it only gains associated analysis information. (Section 5.2)
4. Another desugaring step is defined in STRATEGO that simplifies the AST based on the now available analysis results. Note that not only the AST is transformed. The analysis results are transformed with it to stay valid for the transformed AST. (Section 5.4)

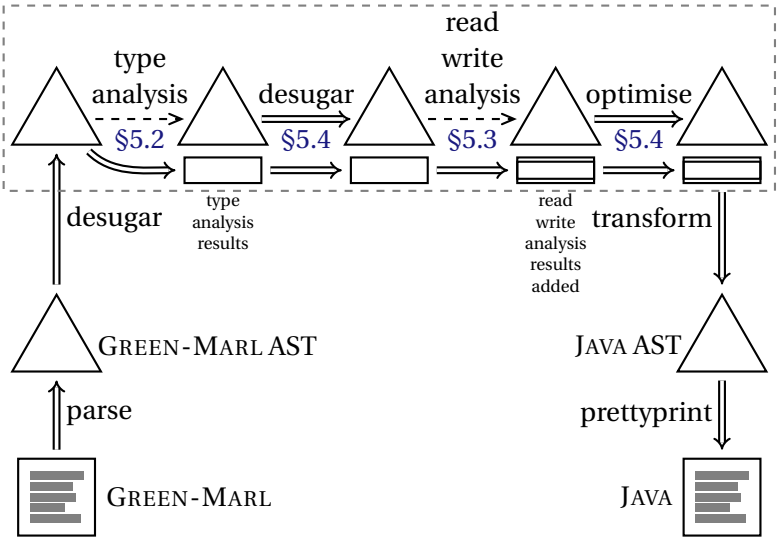


Figure 5.1 – Overview of the `gm_spoofax` compiler steps. The double arrows are transformations. The dashed arrows are identity transformations. The boxes with grey lines are source code. The triangles are trees. The rectangles under the trees are associated information.

5. We apply the *read-write analysis* on the desugared AST with type information. This analysis is defined in STRATEGO. Note that the AST again gains extra analysis information, but the tree itself is not changed. (Section 5.3)
6. The *preservation of analysis results* remains a theme, in the transformation step after the read-write analysis. We apply optimisations based on the analysis results, but the analysis results need to be maintained to allow further steps access to still valid results. We describe our strategy for the preservation, which is again defined in STRATEGO. (Section 5.4)
7. After all optimisation and desugaring steps are complete, the GREEN-MARL AST is transformed into a JAVA AST with STRATEGO.
8. This AST is then turned into JAVA code with a prettyprinter that was derived from an SDF3 JAVA grammar.

5.2 Type System

The formal specification of the type system mixes the name binding and type rules in one formalism. In Spoofax, type rules are conceptually separated from name binding rules. For these two aspects, Spoofax provides name binding language NABL and type system language TS. The argument for a separate name binding language is as follows:

> The references in a language are governed by rules for name binding and scope. The key concepts in these rules are *definitions* that introduce names, *references* to definitions, and *scopes* that restrict the visibility

```
Block(_):
  scopes Variable, Property

Decl(ty@IntTy(), v):
  defines Variable v
    of type ty
  in subsequent scope
```

$$\frac{\Gamma \vdash^{*} s^{*} : \_}{\Gamma \vdash \{s^{*}\}} \quad \text{[block]}$$

$$\frac{\begin{array}{l}\gamma \vdash t_v \Rightarrow \tau \\ \wedge\ \Gamma_v(n) = \bot \\ \wedge\ \Gamma_v[n \mapsto \langle \tau, \mathbf{w} \rangle] = \Gamma'\end{array}}{\tau', \gamma, \Gamma \vdash t_v\ n; : \Gamma'} \quad \text{[decl-v]}$$

Figure 5.2 – NaBL rules (left) and formal rules (right) for block scope and declarations in subsequent scope.

> of definitions. However, those rules are typically not directly expressed. Rather they are programmatically encoded and repeated in many parts of a language implementation, such as the definition of a substitution function, the implementation of name resolution for editor services, and refactorings.
> —Visser et al. [38] (p. 101,102)

The two meta-languages are complementary systems. NaBL rules can refer to types as if the type analysis is already complete, and TS rules can use bindings as if the name analysis is already complete.

The two languages share a concept of properties. Properties are a generalisation of types, and can be attached to names in NaBL rules and to arbitrary AST nodes in TS rules. Because Green-Marl also has a concept of graph properties, we call the properties in the meta-languages NaBL properties, as property names are declared in NaBL.

In the remainder of this section, we show examples of the NaBL and TS rules that easily correspond to the formal rules and discuss the more challenging parts of the implementation.

Block scope and subsequent scope. NaBL has a notion of namespaces and scopes. A namespace is the space to which a name belongs. Green-Marl's namespaces are `Variable`, `Property`, and `Procedure`. When an AST node scopes a certain namespace, the names that are defined within that sub-tree are only visible within this scope. In Green-Marl, block statements scope variables and properties. In rule [block] in Figure 5.2, this is encoded by *not* propagating the environment we received from the sub-statements. In NaBL we declare this with a **scopes** clause.

Local declarations within a block define names for the following statements. In the formal rule [decl-v], we add the name to the environment and return it to be used for the next declarations. In NaBL, we 1) define the variable, 2) declare the type of the variable, and 3) restrict the scope to the subsequent statements. Without the restriction of the scope, the variable would be visible to any statements before the declaration statement in the block. Now, NaBL creates an anonymous scope for the next statements in the list and defines the new name inside this new scope.

Shadowing. Green-Marl does not allow shadowing of local variables. Rule [decl-v] encodes that a variable must not have been defined before. We cannot get the same

```
// disallow any and all shadowing
nabl-constraint(|ctx): Decl(ty,n) → <fail>
where not(PropTy(_,_,_) := ty)
    ; lookup := <nabl-lookup-lexical-parent(|ctx)> n
    ; <task-create-error-on-success(|ctx, lookup
        , $[Shadowing (duplicate) definition])> n

// constrain types to define their associated graph when there are
//  multiple defined
nabl-constraint(|ctx) = ?term; one(?i@Implicit())
  ; <has-annotation(?Use(lookup))> i
  ; <task-create-error-on-multiple(|ctx, lookup
      , $[Multiple graphs found, specify associated graph])> term
  ; fail
```

Figure 5.3 – Stratego rules that add custom NaBL constraints for shadowing and implicit graph parameters in the presence of multiple graphs.

behaviour from inside NABL. By default, NABL assumes names to be unique, so that a second definition of a name in the same scope results in an automatic error message. But NABL also assumes that a sub-scope may override defined names. Therefore we use the custom constraints to disallow shadowing. NABL provides an extension point for such custom constraints in STRATEGO, a term rewrite language. In Figure 5.3 we set an error on a variable declaration when another variable of that name is already defined.

Graph references. Domain-specific types in GREEN-MARL can have a reference to a graph that they belong to. When they do not have this reference, the graph can be inferred if only one graph is in scope. Rule [decl-v] uses semantic type translation to infer or validate the graph reference. For example, to infer the graph of a node, when only one graph is in scope, we use:

$$\langle n \rangle \vdash \mathbf{N} \Rightarrow \mathbf{N}(n) \qquad \text{[n-i]}$$

Name dependent types are not readily supported by NABL and TS. Therefore, we handle the graph reference as a separate NABL property, the *graph* property. Figure 5.4 shows the rules that extract the graph from a type and propagate it to references. The idea is that the declaration of a graph also implicitly declares the graph on a special name. This special name is already present in types that do not specify the graph reference.

We represent the type **node**(g) as `ItemTy(Node(),GraphRef("g"))`. Similarly, the type **node** is represented as `ItemTy(Node(),Implicit())`. In both situations, we need to resolve the graph in order to assign the *graph* property (Figure 5.4, first two lines). Therefore, the NABL rule for input arguments of type **graph** defines not only the graph variable, but also a variable `Implicit()`. Because the implicit graph reference is not in the matched sub-tree, the NABL rule specifies that this is an implicit name definition. Now both kinds of graph references can be resolved. In the TS rule for `Implicit()`, we can look up the definition of the implicit variable.

The separate NABL property for graph references requires a small amount of duplication, since all AST nodes with ***type*** rules also need *graph* rules. This duplication starts to become more cumbersome with parametrised types such as

```
i@Implicit(): refers to Variable i

GraphRef(g): refers to Variable g

FormalInArg(g, ty@GraphTy()):
  defines Variable g
    of type ty
    of varKind InArg()
    of graph g
  implicitly defines Variable Implicit()
    of type ty
    of varKind InArg()
    of graph g
```

```
g@Implicit() has graph g'
where definition of g has graph g'

GraphRef(g) has graph g'
where definition of g has graph g'
  and definition of g : ty
  and ty == GraphTy()
 else error $[expected graph but got [ty]] on g

ItemTy(_, g) has graph g' where g has graph g'
```

Figure 5.4 – NABL rules (top) and TS rules (bottom) for graph references.

`node/edgeProperty` and `map`. Graph properties belong to one graph, but their type parameter may belong to another graph altogether. Maps have a similar problem. A map itself does not belong to any graph, but keys and values may. For the same map, keys and values can belong to different graphs. For example, consider `map(node(g),edge(h))`. The *keygraph* of this map type is `g`, and the *valuegraph* of this map is `h`. We have to use separate properties *propgraph*, *keygraph* and *valuegraph*. And each of these properties need their own duplicate rules.

Type relations. GREEN-MARL has different kinds of related types. The formal rules make use of different semantic sub-domains to constrain rules to certain kinds of types. For example, τ_n in rule [num-op] restricts the arithmetic operations to the numeric types. We model these kinds of types in TS through a *type relation*. Figure 5.5 shows the definition of a `<kind:` relation for the primitive types and the type rule for arithmetic operations.

In the formal specification, we define two rules for `NIL`, one for node and one for edge. The graph name can be any name, and this non-determinism is resolved at the site where this rule is used. In TS we have to be more explicit. We define `NIL` to be of type `NilTy()`, which is a subtype of any node or edge regardless of the graph they belong to (Figure 5.6). We define a subtype relation `<type:` between `a` and `b` where `a` and `b` are either 1) equal under extended equality[1], or 2) when `a` is `NilTy()` and `b` is a node or edge type where we ignore the graph reference. We consistently use this subtype relation instead of direct equality in most other type rules.

[1] Extended equality is another relation we defined where `node` and `node(g)` are equal when only graph `g` is in scope.

```
relations

define <kind:

IntTy()       <kind: NumericKind()
LongTy()      <kind: NumericKind()
FloatTy()     <kind: NumericKind()
DoubleTy()    <kind: NumericKind()

type rules

Mul(e1, e2) +
Div(e1, e2) +
Mod(e1, e2) +
Add(e1, e2) +
Sub(e1, e2) : ty
where e1: ty1 and ty1 <kind: NumericKind()
 else error $[expected numeric type but got [ty1]] on e1
  and e2: ty2 and ty2 <kind: NumericKind()
 else error $[expected numeric type but got [ty2]] on e2
  and ( ty1 <type: ty2 and ty2 ⇒ ty
     or ty2 <type: ty1 and ty1 ⇒ ty )
```

$$\tau_n \quad = \quad \textbf{int} \mid \textbf{long} \mid \textbf{float} \mid \textbf{double}$$

$$\frac{e_1 : \tau_n \ \wedge \ e_2 : \tau_n}{e_1 \ o_n \ e_2 : \tau_n} \qquad \text{[num-op]}$$

Figure 5.5 – Numeric kinds in TS and formal rules (top-right).

```
type rules

NIL() : NilTy()

relations

a <type: b
where a <eq: b
   or   a ⇒ NilTy()
    and ( b ⇒ ItemTy(Node(),_)
       or b ⇒ ItemTy(Edge(),_) )
```

$$\textbf{NIL} : \textbf{N}(n) \qquad \text{[nil-node]}$$

$$\textbf{NIL} : \textbf{E}(n) \qquad \text{[nil-edge]}$$

Figure 5.6 – Type rules for `NIL`, with formal rules on the right.

Return types. Procedures in GREEN-MARL define their return type at the start of the procedure, and use the return statement to return a value of that type. In the formal rules, we passed down the return type of a procedure towards the return statement. In NABL and TS, rules are context-free. Thus, we cannot depend on rules to pass down information. Instead we introduce the artificial reference `Ret()`, to the return statement during the desugaring phase before the analysis. Similar to the implicit graph reference, we define `Ret()` implicitly for each procedure and assign it the type of the return type (Figure 5.7). The TS rules then use this artificial name as a reference to the procedure return type, by looking up the type of the definition of `Ret()`. Alternatively, we could also propagate the return type itself in the transformation as the formal rules do.

Placeholders. In group assignments, we can refer to the current node or edge with a placeholder `_`. This placeholder is treated as just another name in the formal rules, defined within the right-hand side expression [gg]. To type the placeholder, we look up the property to see if it is on nodes or edges.

```
ReturnTy(ty):
  implicitly defines Variable Ret() of type ty

ty@NoReturnTy():
  implicitly defines Variable Ret() of type ty

Return(r@Ret()):
  refers to Variable r

ReturnWith(r@Ret(),_):
  refers to Variable r
```

```
s@Return(r@Ret()) :-
where definition of r : ty
  and ty == NoReturnTy()
 else error $[expected [ty]] on s

ReturnWith(r@Ret(),e) :-
where definition of r : ty
  and e : ety
  and ety <type: ty
 else error $[expected [ty] but got [ety]] on e
```

Figure 5.7 – Name binding rules (top) and the type rules (bottom) for the return type and statement.

$$\frac{\Gamma_v(n_1) = \langle \mathbf{graph}(n_1), \_ \rangle \ \wedge \ \Gamma_g(n_2, n_1) = \langle \mathbf{P}\!<\!\tau_e, \tau_{pc}\!>, \mathbf{w} \rangle \ \wedge \ \Gamma[\_ \mapsto \langle \tau_e, \mathbf{r} \rangle] \vdash e : \tau_{pc}}{\Gamma \vdash n_1 . n_2 = e} \quad [\text{gg}]$$

```
Assign(PropAssign(_,_),_):
  scopes Variable

pa@PropAssign(e,p):
  refers to Property p of graph g
    where e has graph g
  implicitly defines Variable Placeholder()
      of type ty
      of graph g
    where pa has phty ty
      and e has graph g

p@Placeholder(): refers to Variable p
```

```
ph@Placeholder(): ty
where definition of ph : ty

PropAssign(_, p) has phty ItemTy(i1,Ign())
where definition of p : PropTy(i1, ty, _)
  and ( ety == GraphTy()
     or <coll-item> ety ⇒ ItemTy(i2,_) and i1 == i2)
```

Figure 5.8 – Name binding rules (middle) and type rules (bottom) for the group assignment placeholder, and the corresponding formal rule on top.

```
FuncCall(e, "pickRandom", e*) : ItemTy(Node(),Ign())
where e  : ty
  and ty == GraphTy()
 else error $[expected graph but got [ty]] on e
  and e* : [[]]
 else error $[no arguments expected] on e*

FuncCall(e, built-in-pickRandom(), e*) has graph g
where e has graph g
```

Figure 5.9 – The TS rules for `pickRandom`.

In our NABL rules (Figure 5.8), we make assignments scope the variable namespace and implicitly define the placeholder. However, we cannot resolve the property reference and deconstruct the type to give the placeholder the appropriate type from NABL. We can do so in TS, but then we need to get the information back into NABL. So, we create a special NABL property, the placeholder type *phty*. Then we use TS to assign this NABL property to the `PropAssign`, and in NABL we propagate it to the placeholder. The final condition of the TS rule makes sure we only define *phty* when the property assignment is on a graph or a collection of graph elements of the right kind.

Polymorphic functions. In our formal rules, the combination of built-in, overloaded, polymorphic functions are supported through a multi-valued function and an instantiation judgement. TS does not support polymorphism. Instead, we use a separate rule for every built-in function. For example, consider the rules for `pickRandom` in Figure 5.9. We return a node type, with an `Ign()` (ignore) in the graph reference position. We do not compare that AST node in graph element types anyway, as it can hold both an implicit and an explicit graph reference that can refer to the same graph. Instead, we use a separate rule that sets the *graph* NABL property.

5.3 Read-write analysis

The read-write analysis implementation differs strongly from the type system implementation. Whereas our type system implementation was mainly based on declarative meta-languages, which guided the implementation in a particular direction, Spoofax does not provide a specific meta-language for other static analyses. So we used the STRATEGO transformation language, in which we specify a term rewrite system. Rewrite systems are a common implementation approach to static analyses [4].

The read-write analysis is only described by illustration in publications [21,23]. An example GREEN-MARL program is given, along with a table of analysis results and a short description of the results. The only implementation was in `gm_comp` in C++, and the understanding of its details were lost. Therefore our first attempt to understand and recreate this analysis in STRATEGO was a re-engineering effort. Our implementation in `gm_spoofax` is able to provide the same information as the `gm_comp` implementation. However, we achieved this by staying close to the procedural implementation in C++. This means that the STRATEGO code differs strongly from the formal specification.

```
1 read-write-analysis = bottomup(try(set-rwMap(|<rw-analyze>)))
2
3 rw-analyze: Block(s*) → res
4 with rwMaps := <map(get-rwMap);rwMap-unions> s*
5    ; decls  := <retain-all(decl-name);make-set> s*
6    ; no-decl-info := <dict-diff-keys-uri> (rwMaps, decls)
7    ; res := <map-rwInfo(rw-block-helper(|decls))> no-decl-info
```

$$\frac{\Gamma \vdash^{\cup}_{\mathsf{rw}} s^* : \varrho}{\Gamma \vdash_{\mathsf{rw}} \{s^*\} : \varrho_{/\Gamma}} \quad \text{[block]}$$

Figure 5.10 – The bottomup read-write analysis and the analysis rule for block statements (top), and the formal rule for block statements (bottom).

```
rw-iter-helper1(|rng,iter) =
  with(access-pattern := <range-to-AccessPattern> (iter,rng))
; let is-iter = where(\i' → <eq-uri> (iter,i')\)
      iter-prop-to-scalar = Property(is-iter);!Scalar()
      rd2w = \RedDef(_,_) → Write()\
  in RwInfo(try(rd2w), id, !access-pattern, iter-prop-to-scalar)
  end//let
```

$$n_1 \vdash \langle n_2, \mathbf{defer}(n_1), patt \rangle \Rightarrow \langle n_2, \mathbf{write}, patt \rangle \quad \text{[defer]}$$

$$n_1 \vdash \langle n_2, \mathbf{reduce}(n_1, \_), patt \rangle \Rightarrow \langle n_2, \mathbf{write}, patt \rangle \quad \text{[reduce]}$$

$$n_1, patt \vdash \langle n_2, mode, n_1 \rangle \Rightarrow \langle n_2, mode, patt \rangle \quad \text{[name]}$$

Figure 5.11 – The helper rule that replaces a property access through an iterator by the access pattern of the loop/search.

Despite this, we did gain a better understanding of the analysis, which eventually turned into the formal specification we presented earlier. In the rest of this section, we connect some small examples of our STRATEGO code with the formal rules.

Top-level and Block rule. We use an `rw-analyze` rule for every kind of statement, and cache the read-write information in an annotation on the statement. With a generic `bottomup` strategy we apply the analysis on the entire program. Figure 5.10 shows this top-level definition of the read-write analysis, and an example rule for the block statement. The block statement rule gets and combines the read-write information of the statements on Line 4, collects all local declarations (5), removes those local declarations (6), and uses a helper strategy to turn property accesses through those local declarations into random accesses (7). The formal rule applies step 1 in the premise, and steps 2–4 with the restriction to the environment.

The rules for statements are all fairly similar to the formal rules, although more verbose, as demonstrated in the figure. The STRATEGO rules that implement the read-write set transformations are more complex. This is partly due to our incomplete understanding at the time, and partly because these transformations do more than the ones in the formal specification. In this implementation, we migrated the entire read-write analysis from the C++ implementation, including such things as conditionality tracking, collection mutation, and a special property access pattern for breadth-first search iterators. In Figure 5.11 we show the simplest rule, which

replaces a property access with an iterator into a property access with the pattern of the iterator. The `range-to-AccessPattern` strategy corresponds to the $\cdot \vdash_{ap} \cdot \Rightarrow \cdot$ judgement. The `rd2w` strategy corresponds to rules [defer] and [reduce]. And the strategies `!access-pattern, iter-prop-to-scalar` together correspond to [name].

5.4 Preservation of analysis results

The `gm_spoofax` compiler has to do multiple successive transformation steps, e.g. for desugaring and optimisation. Most of these steps require analysis results to inform them. Each step changes the program slightly, but the analysis results were based on the unchanged program. So each step invalidates parts of the analysis results, and we need accurate analysis results to inform the next step.

Fortunately, GREEN-MARL requires certain analyses and transformation that combine well. We can apply incremental updates with every transformation to keep the program well-analysed. In the remainder of this section, we describe our approach to these incremental updates for both of the analyses. Our approach here is language-specific. We discuss possible directions to solving this problem generically in Chapter 7.

Type analysis. When the task engine has finished the type analysis, we can use the analysis results inside STRATEGO to inform program transformations. However, when such a transformation is done we do not automatically gain new name and type information on the transformed program. We cannot naively re-analyse the entire program. The incremental task engine was designed to support incremental changes to the program by the user, not small successive changes by STRATEGO.

However, the transformations we need have a number of properties that keep the changes local to the transformation site: (1) We do not change the types or other NABL properties of existing names. (2) We do not change the existing bindings. (3) We introduce new declarations and references at the same time. (4) When we replace expressions, they are replaced with an expression of the same type. (5) We replace statements, but these do not have types.

Because of these properties of the transformations, we can apply a transformation and be sure that no analysis information in other parts of the AST are invalidated. The part that we transform is the part that we control, therefore we can manually annotate new AST nodes with the correct NABL properties and their values.

This manual annotation is a re-implementation of the name binding and type rules. This is an unfortunate duplication of effort that leaves the transformation code bloated. Making changes to name binding and type rules is also more error-prone with the duplication.

We demonstrate what a transformation with manual updates looks like in Figure 5.12. This is a desugaring from a group assignment to a parallel loop. We need a new name for the iterator, and the range of the loop. The loop introduces a new scope within which the name is contained, so it cannot affect anything outside of the transformed portion of the program. The transformation selects assignments (Line 2) to properties (3), where the type (5) and graph (4) of the property are defined and the assignment is to a graph or collection (6). It then creates the range for the loop from the graph or collection (2) and graph element kind `elemKind`. The `varKind` is

```
1  group-assignment-to-foreach: Assign(pa@PropAssign(ref,propName),expr) →
2    ForEach(Parallel(),IterBounds(nIter,range,NoFilter()),Assign(pa',expr'))
3  where PropTy(elemKind,_,_) := <get-type> propName
4      ; graph   := <get-graph> propName
5      ; reftype := <get-type> ref
6      ; <?GraphTy() + ?CollTy(_,_,_) + ?HOCollTy(_,_)> reftype
7   with (range,varKind) := <group-assign-range> (ref, reftype, elemKind)
8      ; annos := [ (Type(),ItemTy(elemKind,GraphRef(graph)))
9                 , (NablProp_varKind(), varKind)
10                , (NablProp_graph(), graph) ]
11     ; nIter :=
12         <build-group-prefixed-name;add-annotations(|annos)> propName
13     ; rIter := <set-annos> (VarRef(nIter), annos)
14     ; expr' := <alltd(\Placeholder() → rIter\)> expr
15     ; pa'   := <PropAssign(!rIter,id)> pa
```

```
   group-assignment-to-foreach: Assign(pa@PropAssign(ref,propName),expr) →
     ForEach(Parallel(),IterBounds(nIter,range,NoFilter()),Assign(pa',expr'))
3  where PropTy(elemKind,_,_) := <get-type> propName
       ; reftype := <get-type> ref
       ; <?GraphTy() + ?CollTy(_,_,_) + ?HOCollTy(_,_)> reftype
6   with range := <group-assign-range> (ref, reftype, elemKind)
       ; nIter := <newname> propName
       ; expr' := <alltd(\Placeholder() → VarRef(nIter)\)> expr
9      ; pa'   := <PropAssign(!VarRef(nIter),id)> pa
```

Figure 5.12 – Transformation rule for turning group assignment into a parallel loop with (top) and without (bottom) code to update the analysis information.

the type of iterator that the range yields and is kept in a separate NABL property (8). The name for the iterator is created (12), and both the definition and reference (13) are annotated with the properties that we have gathered. We replace the placeholder and property assignment left-hand side with the iterator references (15), and use the adapted expression and property assignment in the final result (2).

In the same figure, we also give the program with the analysis update code stripped from it. This shows that for this transformation rule, the incremental update requires a 67% increase in lines of code.

Read-write analysis. The read-write analysis is not based on the task engine. Because we wrote it directly in STRATEGO we have more control over when and where we apply it. More importantly, it is a bottom-up analysis, at most the spine towards the root of the AST has to be changed when we do a transformation somewhere in the tree. And in fact, the transformations we apply do not affect the abstract read-write information of the names that are subject to the transformation. We only introduce new names inside new scopes, therefore those names are abstracted over in the parent nodes of the transformed node.

We wrote our STRATEGO implementation so we can reapply the analysis on a transformed AST where it reuses the results that are still cached on unchanged subtrees. This makes sure that we do not have to manually apply the read-write rules in the transformation, as we have to do with the NABL and TS rules to preserve a completely analysed program.

Chapter 6

Related Work

In this chapter we describe the previous work on related subjects, namely GREEN-MARL, formal static semantics of programming languages, language workbenches, and static analyses.

6.1 GREEN-MARL

The GREEN-MARL graph analysis language was originally developed as part of the PhD dissertation of Sungpack Hong [21]. Parts of this work were also published [23]. Now, the GREEN-MARL language is defined in a Language Specification. We formally described the language based on version 0.6.2 [22]. The latest version that is online at this time is 0.7.1 [24].

The language definition describes the GREEN-MARL's purpose: easy development of graph-data processing programs. It is intended to exploit modern, parallel hardware. It does so by offering the user features to specify parallelism in algorithm implementations. The language syntax is first described, with a formal BNF-like notation and accompanying text that describes identifiers, literals, and comments. Then, through prose and examples, the basic language entities are described (procedures, statements and expressions), and a short description of the scoping rules is given. This description using prose text and examples continues as the type system, parallel execution semantics, and miscellaneous details are described.

The dissertation and publication go into more detail and describe the compilation process of GREEN-MARL to a parallel, shared-memory environment and to a distributed environment. Checks and optimisations are described, again informally with prose text and examples. A particular focus is the experimental evidence of GREEN-MARL's efficiency, where its simple algorithm implementations are faster than manually optimised implementations in other programming languages. Our work does not focus on the efficiency of programs compiled in the language. Instead we describe the language in a structured, formal way.

6.2 Formal Static semantics

Specification of the formal static semantics of programming languages is a practice with a long history. A prominent example is the definition of STANDARD ML [30],

which describes that the design of the language, the formal definition, and the implementation all influenced each other, and they cannot imagine that each of them could have been properly when done separately.

Other languages, such as JAVA, gained a formal static semantics outside of the definition. As a result, there are many lightweight versions of JAVA where the language is reduced to enable rigorous arguments [12,16,34]. One example of such a static semantics on a particularly small core is FEATHERWEIGHT JAVA [26], which is described as: 'Featherweight Java bears a similar relation to Java as the lambda-calculus does to languages such as ML and Haskell.' (p. 396) This example only formally describes the static semantics of a small core of JAVA, but it still found a bug in the Generic JAVA compiler [10] (the precursor of generics in JAVA 1.5). And, the authors argue, it has been a useful tool clarify their thought.

We have experienced our formalisation efforts similarly. Describing the static semantics of GREEN-MARL has added rigour to our understanding of the language, and led us to ask after points where the language specification was unclear.

And yet, not all language designers are proponents of formally defined semantics of programming languages. Hudak et al. argue that for HASKELL, the absence of a formal definition allowed the language to evolve more easily, as the cost of updating the formal specification of the language with every proposed change would be too heavy and would discourage all changes [25] (p. 9).

We certainly understand the argument. In our experience, it is difficult to develop a complete and correct formal specification of a programming language when it is all written by hand. We imagine that automatic theorem provers could help the development of both the first specification and any updates. We find it even more promising that an executable, declarative meta-language such as the meta-languages in Spoofax can derive formal specifications automatically [38]. We do believe there is merit in the definition of a formal static semantics to clarify the meaning of the programming language, regardless of whether that semantics is written, defined in a theorem prover or derived from a declarative specification.

6.3 Language workbenches

We used the Spoofax language workbench for `gm_spoofax`, but there are many other language workbenches [13]. We will focus on the following modern ones: XText [15], MPS [27], and RASCAL [28]. Although RASCAL is not language workbench so much as a meta-programming language, it is still capable of defining domain-specific languages.

Xtext [15] is a workbench for developing programming languages and is developed as part of the Eclipse Modeling Framework project. It provides JAVA-like language Xtend in which a type system can be implemented through low-level calculations, but it also provides Xbase, Xsemantics [8] and XTS [40]. The comparison between these options [9] shows that Xtend has the verbosity of a plain JAVA solution but can be used to implement any type system. Xbase is well-suited for JAVA-like languages that need tight integration with JAVA. Xbase gives the domain-specific language the complete JAVA type system, which is very helpful if you need the JAVA type system, but not otherwise. Xsemantics is an interesting domain-specific language for definition

of both static and dynamic semantics that uses a syntax that is similar to formal inference rules. This sounds like a promising way to implement the static semantics of a programming language with virtually no gap between the specification and implementation. We do wonder whether this system is truly capable of supporting all rules that we use. If so, we might also use it to implement the read-write analysis of GREEN-MARL. XTS is a domain-specific language specifically for type systems and is therefore more concise than Xsemantics. However, it is also more specialised towards standard object-oriented type system features, therefore the implementation of GREEN-MARL's type system would likely still involve a lot of JAVA code, connected with XTS rules for simpler parts of the type system. Although XTS is a JAVA library with domain-specific language, we expect that even the library will not be able to support GREEN-MARL's name-dependent types, therefore we cannot predict the feasibility of implementing GREEN-MARL's type system in XTS.

MPS [27] is a language workbench with a distinctive projectional editor. The type system engine of the language workbench uses unification and allows the language implementer to define type rules in the form of equations that should be solved [41]. This unification approach looks quite powerful, but we cannot ascertain how much name and type information can be mixed in these equations. Even based on the in-depth documentation[1], we do not see features that would be able to model the name-dependent types of GREEN-MARL. What we can see is that MPS' built-in support for data-flow analysis is not able to support GREEN-MARL's tree-based, symbolic read-write analysis. The built-in support for data-flow analysis requires a translation of the domain-specific language to a simple intermediate representation that consists of reads, writes, labels, jumps, and subroutines. It resembles a tiny assembly language that abstracts over operations and only models sequential flow of control.

RASCAL [28] is a domain-specific language for program analysis and transformation. It has a dedicated location type with literals, which is heavily used in its library for code analysis M^3 [5], and a domain-specific language and library DCFlow for constructing control-flow graphs [20]. RASCAL has the term rewriting and generic traversals that we know from STRATEGO but is also statically typed, supports types such as sets, relations and locations, and has libraries tailored to static analyses. Based on this, we consider the language very well-suited for a cleaner implementation of the read-write analysis of GREEN-MARL. However, for the type system of GREEN-MARL, RASCAL does not seem to offer anything close to the high-level specification of TS or Xsemantics. A lower-level, more operational implementation of the type system is certainly feasible, and likely to be less verbose than an implementation in JAVA, but would still be quite involved.

6.4 Dependence analyses

The read-write analysis of GREEN-MARL is a form of dependence analysis. Or rather, it gathers all knowledge that is required to calculate the dependence between statements by simple set intersection. Dependence analysis is a well-established analysis, there is even a full book on Optimising Compilers for Modern Architectures based

[1] https://confluence.jetbrains.com/display/MPSD30/Typesystem

on dependence analysis [3]. The difference between the dependence analysis from this book and that of GREEN-MARL is that symbolic analysis is not the focus of the book. It is mentioned in small subsections in two places in the book, and most of the work uses examples based on constant loop bounds. GREEN-MARL's read-write analysis on the other hand is completely symbolic, and abstracts over sub-statement dependences.

Chapter 7

Discussion

In this chapter we discuss the insights from this thesis, the remaining challenges, and future work. We structure the chapter by our research questions.

RQ1. What is the static semantics of GREEN-MARL?

The type system chapter of the GREEN-MARL Language Specification [22] starts with the sentence: 'Green-Marl has a very simple type system.' The formalisation in this thesis tells a different story. We have standard type system features such as parametrised types and overloading. We also have domain specific types that are name-dependent and use a limited form of type inference. For each of these features we looked for the underlying general principle from standard type theory, and how we could combine it with the theory we needed for other features. Chapter 3 answers this research question in full.

Interesting future work includes an extension of the type system to update to GREEN-MARL version 0.7 [24]. That version introduces vector types. These vector types are parametrised by type and by length, which makes them value-dependent types.

RQ2. How can the static semantics be declared in Spoofax?

Spoofax is language designer's workbench, and aspires to be 'a one-stop-shop for development, implementation, and validation of language designs' [38]. We have found that this aspiration is not yet attained, given the gap between our formal rules for GREEN-MARL and our implementation. Spoofax's meta-languages for the specification of name binding and type rules—NABL and TS—do provide us with a way to declare the static semantics of GREEN-MARL, but they sometimes require inelegant workarounds:

Two workarounds. One challenge is the inability for NABL and TS to handle return types of procedures declaratively. Instead we apply rewrite rules before the analysis phase. These rules propagate an artificial name to return statements to connect the procedure declaration with the return statements. The formal rules simply pass down the return type from the procedure declaration to the return statements, which is not possible in NABL/TS.

Another challenge comes from the name dependent types of GREEN-MARL, where a type `node(g)` refers to graph `g` to which it belongs. We model the graph reference in types as a separate NABL property. But this does not scale well to parametrised types, where the type arguments can have different graph references. That type needs two NABL properties to model the names in the name-dependent types. Each of those properties needs duplicate rules to propagate the graph references.

Scope graphs and constraints. We expect a more declarative approach within the Spoofax ecosystem to come from the new Scope Graph [32] and type constraints [36] work. The mix between names and types is an explicit topic in the latter work on type constraints. We have done a preliminary investigation and are confident that this approach can model GREEN-MARL's type system.

For the return type situation, we could use a transformation to a scope graph that mimics the formal rules, simply passing down the return type. However, this hides the connection between the procedure return type and the return statement. If instead of top-down passing of information, we find a way to put that connection into the constraints during a bottom-up traversal of the program, that would help with incremental analysis [14].

RQ3. What is the formal semantics of the dependence analysis of GREEN-MARL?

The dependence analysis of GREEN-MARL, called the read-write analysis, is mentioned in publications [23], but not described in detail. Only examples with analysis results are shown, where the meaning and use of the analysis results are explained. Based on this information and the original implementation[1] we reconstructed the analysis.

We give a formal specification of the dependence analysis based on inference rules. We purposely removed some of the features from the analysis to make the formalisation easier to understand. Future work is a description of the extensions that restore the original capabilities of the implementation.

More work that could be done in the future is the formalisation of additional analyses from the `gm_comp` compiler. These can range from classic analyses, such as Reaching Definitions, to language specific ones, such as finding nested breadth-first searches, which are currently not supported and should therefore be statically disallowed. A number of these smaller analyses are implemented but not formalised.

RQ4. How can this dependence analysis be declared in Spoofax?

Spoofax has a term rewrite language called STRATEGO. Our compiler implementation uses STRATEGO's rewrite rules to implement the read-write analysis. The current

[1] Old open-source version: https://github.com/stanford-ppl/Green-Marl/blob/4c0d62e67d431d535ca27140df60b25c234a808b/src/frontend/gm_rw_analysis.cc

implementation is not closely related to the formal rules. We believe that a new implementation in STRATEGO can come closer to the formal specification. However, this does not solve how much more verbose STRATEGO is. A more domain-specific meta-language for static analysis would be more appropriate, but since Spoofax lacks such a meta-language, this is a future research topic.

The design goal of this meta-language would be a declarative approach to static analyses. The ability to influence and be influenced by already available analyses for name binding and types is an interesting feature. Scope graphs are quite powerful because they can have this interaction with types. For this feature, the meta-language should target a constraint based system. The Monotone Framework[33] for example, is an interesting meta-language basis that could interact with constraint based scope graphs and types.

Analysis driven optimisation. We only apply a loop merging optimisation to loops that are consecutive in the program. Since not all loops are actually adjacent in the program, we move all loops upwards as far as the dependences allow us, to create more optimisation opportunities. This is a heuristic from `gm_comp` and can be counter-productive in some cases.

Instead of this heuristic, we could think of this problem as a control-flow graph, where we relax the strictly linear flow in a block of statements to the graph of dependences that are found between the statements. With this graph, we should be able to easily identify loops that do not have statements in between.

We did not implement this idea because time was short and STRATEGO is oriented towards trees rather than graphs. An analysis driven specification language for optimisations is interesting future work.

RQ5. How can analysis results be kept consistent after transformations?

A recurring issue we found while implementing GREEN-MARL in Spoofax is the need to manually update analysis information when applying transformations. Whether these transformations were used for desugaring or optimisation, any change to the abstract syntax tree could break the connection to the analysis results or invalidate them.

Our approach is to manually patch the name and type information in our transformations. We could not use the incremental task engine that backs NABL and TS. That engine is meant for incremental changes by the user, which occur less frequently than small transformation steps in the compiler.

Our STRATEGO implementation of the read-write analysis gave us much more control over its evaluation. We used this control to make use of earlier analysis results if those were available. This allows partial reuse of analysis results on transformed programs, and prevents the analysis from polluting the transformations.

Our manual updates seems mechanical and very similar to the NABL and TS rules. We suspect that for an easily analysable subset of STRATEGO, we could automate the patching of analysis results. With a more declarative approach to the

read-write analysis, we consider this feasible for that analysis as well. This automated analysis consistency for intra-language transformations is another interesting research avenue.

Bibliography

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. doi: 10.1109/IEEESTD.2008.4610935. 21

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983. ISBN 0-201-00023-7. 5

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001. ISBN 1-55860-286-0. 68

[4] U. Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In T. Gyimóthy, editor, *Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 121–135. Springer Berlin Heidelberg, 1996. ISBN 978-3-540-61053-3. doi: 10.1007/3-540-61053-7_57. URL http://dx.doi.org/10.1007/3-540-61053-7_57. 2, 60

[5] B. Basten, M. H. 0001, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, and J. J. Vinju. M3: A general model for code analytics in rascal. In O. Baysal and L. Guerrouj, editors, *1st IEEE International Workshop on Software Analytics, SWAN 2015, Montreal, QC, Canada, March 2, 2015*, pages 25–28. IEEE, 2015. doi: http://dx.doi.org/10.1109/SWAN.2015.7070485. 67

[6] A. Bavelas. Communication patterns in task-oriented groups. *Journal of the acoustical society of America*, 1950. 5

[7] R. Bellman. On a routing problem. Technical report, DTIC Document, 1956. 5, 7

[8] L. Bettini. Implementing Java-like languages in Xtext with Xsemantics. In S. Y. Shin and J. C. Maldonado, editors, *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 1559–1564. ACM, 2013. ISBN 978-1-4503-1656-9. doi: http://doi.acm.org/10.1145/2480362.2480654. 66

[9] L. Bettini, D. Stoll, M. Völter, and S. Colameo. Approaches and tools for implementing type systems in Xtext. In K. Czarnecki and G. Hedin, editors, *Software*

Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers, volume 7745 of *Lecture Notes in Computer Science*, pages 392–412. Springer, 2012. ISBN 978-3-642-36089-3. doi: http://dx.doi.org/10.1007/978-3-642-36089-3_22. 66

[10] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *OOPSLA*, pages 183–200, 1998. doi: http://doi.acm.org/10.1145/286936.286957. 66

[11] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008. doi: http://dx.doi.org/10.1016/j.scico.2007.11.003. 53

[12] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the java type system sound? *TAPOS*, 5(1):3–24, 1999. 66

[13] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. The state of the art in language workbenches - Conclusions from the language workbench challenge. In M. Erwig, R. F. Paige, and E. V. Wyk, editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer, 2013. ISBN 978-3-319-02653-4. doi: http://dx.doi.org/10.1007/978-3-319-02654-1_11. 66

[14] S. Erdweg, O. Bracevac, E. Kuci, M. Krebs, and M. Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In J. Aldrich and P. Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 880–897. ACM, 2015. ISBN 978-1-4503-3689-5. doi: http://doi.acm.org/10.1145/2814270.2814277. 70

[15] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010. 1, 66

[16] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183, New York, NY, USA, 1998. ACM. doi: http://doi.acm.org/10.1145/268946.268961. 66

[17] L. R. Ford Jr. Network flow theory. Technical report, DTIC Document, 1956. 5, 7

[18] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005. 1

[19] L. C. Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1979. 5

[20] M. Hills. Streamlining control flow graph construction with DCFlow. In B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju, editors, *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *Lecture Notes in Computer Science*, pages 322–341. Springer, 2014. ISBN 978-3-319-11244-2. doi: http://dx.doi.org/10.1007/978-3-319-11245-9_18. 67

[21] S. Hong. *Green-Marl: A Domain Specific Language for Graph Analysis*. PhD thesis, Stanford University, 2013. 60, 65

[22] S. Hong and M. Sevenich. *The Green-Marl Language Specification*, 0.6.2 edition, December 2014. Archived version: https://web.archive.org/web/20150319014607/http://docs.oracle.com/cd/E56133_01/Green_Marl_Language_Specification.pdf. 2, 65, 69

[23] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In T. Harris and M. L. Scott, editors, *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 349–362. ACM, 2012. ISBN 978-1-4503-0759-8. doi: http://doi.acm.org/10.1145/2150976.2151013. 1, 2, 11, 60, 65, 70

[24] S. Hong, M. Sevenich, and J. Smits. *The Green-Marl Language Specification*, 0.7.1 edition, June 2015. Archived version: https://web.archive.org/web/20160129153812/http://docs.oracle.com/cd/E56133_01/Green_Marl_Language_Specification.pdf. 65, 69

[25] P. Hudak, J. Hughes, S. L. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In B. G. Ryder and B. Hailpern, editors, *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, pages 1–55. ACM, 2007. doi: http://doi.acm.org/10.1145/1238844.1238856. 66

[26] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001. doi: http://doi.acm.org/10.1145/503502.503505. 66

[27] JetBrains. Meta programming system. https://www.jetbrains.com/mps. 1, 66, 67

[28] P. Klint, T. van der Storm, and J. J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pages 168–177. IEEE Computer Society, 2009. ISBN 978-0-7695-3793-1. doi: http://doi.ieeecomputersociety.org/10.1109/SCAM.2009.28. 66, 67

[29] G. D. P. Konat, L. C. L. Kats, G. Wachsmuth, and E. Visser. Declarative name binding and scope rules. In K. Czarnecki and G. Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012. ISBN 978-3-642-36089-3. doi: http://dx.doi.org/10.1007/978-3-642-36089-3_18. 2, 53

[30] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML, Revised*. MIT Press, Cambridge, MA, USA, 1997. 65

[31] P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, 2009. doi: http://dx.doi.org/10.1016/j.entcs.2009.07.073. 18

[32] P. Neron, A. P. Tolmach, E. Visser, and G. Wachsmuth. A theory of name resolution. In J. Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. ISBN 978-3-662-46668-1. doi: http://dx.doi.org/10.1007/978-3-662-46669-8_9. 70

[33] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2005. ISBN 978-3-540-65410-0. doi: http://www.springer.com/computer/theoretical+computer+science/book/978-3-540-65410-0. 71

[34] T. Nipkow and D. von Oheimb. Java$_{light}$ is type-safe — definitely. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 161–170, New York, NY, USA, 1998. ACM. doi: http://doi.acm.org/10.1145/268946.268960. 66

[35] Oracle Corporation. PGX 1.1.0 Documentation – Closeness Centrality. https://docs.oracle.com/cd/E56133_01/reference/algorithms/closeness_centrality.html, 2015. URL https://docs.oracle.com/cd/E56133_01/reference/algorithms/closeness_centrality.html. Accessed on 2015-08-30. 6, 8

[36] H. van Antwerpen, P. Neron, A. P. Tolmach, E. Visser, and G. Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *PEPM*, pages 49–60, January 2016. doi: http://dx.doi.org/10.1145/2847538.2847543. 70

[37] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000. doi: http://doi.acm.org/10.1145/352029.352035. 1

[38] E. Visser, G. Wachsmuth, A. P. Tolmach, P. Neron, V. A. Vergu, A. Passalaqua, and G. D. P. Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In A. P. Black, S. Krishnamurthi, B. Bruegge, and J. N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and*

Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014, pages 95–111. ACM, 2014. ISBN 978-1-4503-3210-1. doi: http://doi.acm.org/10.1145/2661136.2661149. 53, 55, 66, 69

[39] T. Vollebregt, L. C. L. Kats, and E. Visser. Declarative specification of template-based textual editors. In A. Sloane and S. Andova, editors, *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*, page 8. ACM, 2012. ISBN 978-1-4503-1536-4. doi: http://doi.acm.org/10.1145/2427048.2427056. 53

[40] M. Völter. Xtext/TS - a typesystem framwork for Xtext. February 2011. URL http://www.infoq.com/articles/xtext_ts. 66

[41] M. Völter. Language and ide modularization and composition with mps. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers*, volume 7680 of *Lecture Notes in Computer Science*, pages 383–430. Springer, 2011. ISBN 978-3-642-35992-7. doi: http://dx.doi.org/10.1007/978-3-642-35992-7_11. 67

[42] G. Wachsmuth, G. D. P. Konat, V. A. Vergu, D. M. Groenewegen, and E. Visser. A language independent task engine for incremental name and type analysis. In M. Erwig, R. F. Paige, and E. V. Wyk, editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 260–280. Springer, 2013. ISBN 978-3-319-02653-4. doi: http://dx.doi.org/10.1007/978-3-319-02654-1_15. 53

[43] G. Wachsmuth, G. D. P. Konat, and E. Visser. Language design with the spoofax language workbench. *IEEE Software*, 31(5):35–43, 2014. doi: http://dx.doi.org/10.1109/MS.2014.100. 1, 53

Appendix A

Type system overview figures

This appendix contains summary figures of GREEN-MARL's judgements, syntax, semantic domains, and semantic type translation.

judgements

$\Gamma \vdash e : \tau$	well-typed expressions
$\tau, \gamma, \Gamma \vdash s$	well-formed statements
$\alpha, \Gamma \vdash e_r : \tau$	well-typed references
$\tau, \gamma, \Gamma \vdash s : \Gamma'$	variable and graph declarations
$\gamma \vdash t \Rightarrow \tau$	semantic type translation
$\iota, \Gamma \vdash i : \Gamma'$	traversal iterators
$\Gamma \vdash i : \Gamma'$	loop iterators
$\Gamma \vdash r : \tau$	well-typed ranges
$\vdash u$	well-formed units
$\Gamma \vdash p : \Gamma'$	procedure declarations
$\Gamma \vdash p$	well-formed procedure declarations
$\alpha, \gamma \vdash f : \tau$	well-typed formal arguments
$\gamma \vdash f : \gamma'$	formal graph arguments
$\alpha, \gamma, \Gamma \vdash f : \Gamma'$	formal argument names
$\Gamma \vdash a : \tau$	well-typed output arguments
$\gamma \vdash t \Rightarrow \tau$	translation of syntactic to semantic types
$\vdash \tau \Rightarrow n$	graph reference extraction

Figure A.1 – GREEN-MARL judgements.

syntax

u	=	p^*	compilation units
p	=	**proc** n (**(**f^* **;** f^***)**: t **{**s^***}**	procedure declarations
	|	**proc** n (**(**f^* **;** f^***){**s^***}**	
f	=	n **:** t	formal arguments
s	=	**{**s^***}**	block statements
	|	$t\,n$**;**	local declaration
	|	n**(**e^* **;** a^***);**	procedure calls
	|	e**.**n**(**e^***);**	function calls
	|	**return** e**;** | **return;**	return statements
	|	**if(**e**)** s **else** s	conditional statements
	|	**while(**e**)** s | **do** s **while(**e**);**	
	|	**for** i s	sequential iteration
	|	**foreach** i s	parallel iteration
	|	**inDFS** i_s s **inPost(**e**)** s	depth-first searches
	|	**inBFS** i_s s **inReverse(**e**)** s	breadth-first searches
	|	e_r **<=** e | e_r **<=** e **@** n	deferred assignments
	|	e_r ra_n e | e_r ra_l e | e_r **<**e_r^***>** ra_c e **<**e^***>**	reduction assignments
i	=	**(**n**:**r**)(**e**)**	iterator declarations
i_s	=	**(**n**:** n**.nodes from** n**)(**e**)[**e**]**	
	|	**(**n**:** n**^.nodes from** n**)(**e**)[**e**]**	search iterators
r	=	n**.nodes** | n**.edges** | n**^.edges**	graph ranges
	|	n**.inNbrs** | n**.outNbrs**	node ranges
	|	n**.inEdges** | n**.outEdges**	
	|	n**.upNbrs** | n**.downNbrs**	BFS iterator ranges
	|	n**.upEdges** | n**.downEdges**	
	|	n**.items** | n**^.items**	collection ranges
ra_n	=	**+=** | ***=** | ra_c	numeric reduction assignments
ra_c	=	**max=** | **min=**	comparison reduction assignments
ra_l	=	**&=** | **|=**	logic reduction assigments
e	=	e_r	references
	|	n**(**e^* **;** a^***)**	procedure calls
	|	e**.**n **(**e^* **)**	function calls
	|	**+INF** | **−INF** | l_i | l_f	numeric literals
	|	l_s	string and date literals
	|	**true** | **false**	boolean literals
	|	**NIL**	node/edge literal
	|	$-\,e$ | **|**e**|** | e o_n e | **(**t**)** e	numeric expressions
	|	e o_c e | **!**e | e o_l e | e **?**e **:**e	boolean expressions
	|	ro_n i **{**e **}** | ro_l i **{**e **}**	reduction expressions
o_n	=	**+** | **−** | ***** | **/** | **%**	numeric operators
o_c	=	**<** | **<=** | **>=** | **>** | o_e	comparison operators
o_e	=	**==** | **!=**	equality operators
o_l	=	**&&** | **||**	logic operators
ro_n	=	**sum** | **product** | **max** | **min**	numeric reduction operators
ro_l	=	**any** | **all**	logic reduction operators
a	=	e_r | a_i	arguments
a_i	=	**#**	ignored arguments

n	procedure, function, variable, property names
l_i	integer literals
l_f	floating point literals
l_s	string and date literals

Figure A.2 – GREEN-MARL values.

syntax

t	=	t_v	variable types
	|	t_g	graph property types
t_v	=	**graph**	graph type
	|	t_p	primitive types
	|	t_m	map types
	|	t_e	graph elements
	|	$t_c \mid t_{cc}$	collections
t_m	=	**map**<t, t>	
t_g	=	**N_P**<t_d> | **N_P**<t_d>(n)	node properties
	|	**E_P**<t_d> | **E_P**<t_d>(n)	edge properties
t_d	=	$t_p \mid t_c$	properties destinations
t_p	=	t_n	numeric types
	|	**string** | **date**	string and date types
	|	**bool**	boolean type
t_n	=	**int** | **long** | **float** | **double**	
t_e	=	**N** | **N**(n) | **E** | **E**(n)	graph elements
t_c	=	**N_S** | **N_S**(n) | **E_S** | **E_S**(n)	sets
	|	**N_Q** | **N_Q**(n) | **E_Q** | **E_Q**(n)	sequences
	|	**N_O** | **N_O**(n) | **E_O** | **E_O**(n)	orders
t_{cc}	=	**collection**<t_c>	collections

semantic domains

γ	=	n^*	graph arguments
α	=	**r** | **w**	access context
ι	=	**s** | **b** | **c** | **n**	iterator context
$\hat{\tau}$			type variables
τ	=	**void** | **graph**(n) | τ_p	semantic types
	|	$\tau_m \mid \tau_e \mid \tau_c \mid \tau_{cc} \mid \tau_g \mid \tau_i$	
τ_p	=	t_p	
τ_n	=	t_n	
τ_m	=	**map**<τ, τ>	maps
τ_e	=	**N**(n) | **E**(n)	graph elements
τ_c	=	**S**<τ_e> | **Q**<τ_e> | **O**<τ_e>	graph collections
τ_{cc}	=	**Q**<τ_c>	collections
τ_g	=	**P**<τ_e, τ>	graph properties
τ_f	=	**F**<τ, τ^*, τ>	function type
τ_p	=	**P**<τ^*, τ^*, τ>	procedure type
τ_i	=	**I**<ι, τ_e>	iterator
$\check{\tau}$			types with type variables
σ	=	$\tau \mid \sigma_f \mid \sigma_p$	semantic type schemes
σ_f	=	$\forall\ n^*, \hat{\tau}^*.\ \check{\tau}_f$	function signatures
σ_p	=	$\forall\ n^*, \hat{\tau}^*.\ \check{\tau}_p$	procedure signatures
Γ	=	$\Gamma_v \times \Gamma_g \times \Gamma_p$	all environments
Γ_v	=	$n \rightarrow_{fin} (\tau \times \alpha)$	variables
Γ_g	=	$n \rightarrow_{fin} n \rightarrow_{fin} (\tau \times \alpha)$	graph properties
Γ_p	=	$n \rightarrow_{fin} \sigma_p$	procedures
Γ_f	=	$n \rightarrow_{fin} \sigma_f$	built-in functions

Figure A.3 – GREEN-MARL types and semantic domains.

semantic type translation $\boxed{\gamma \vdash t \Rightarrow \tau}$

$$\gamma \vdash t_p \Rightarrow t_p \quad \text{[sem-pt]}$$

$$\langle n \rangle \vdash \mathtt{N} \Rightarrow \mathtt{N}(n) \quad \text{[n-i]}$$

$$\langle n \rangle \vdash \mathtt{E} \Rightarrow \mathtt{E}(n) \quad \text{[e-i]}$$

$$\langle \ldots, n, \ldots \rangle \vdash \mathtt{N}(n) \Rightarrow \mathtt{N}(n) \quad \text{[sem-n]}$$

$$\langle \ldots, n, \ldots \rangle \vdash \mathtt{E}(n) \Rightarrow \mathtt{E}(n) \quad \text{[sem-e]}$$

$$\langle n \rangle \vdash \mathtt{N\_S} \Rightarrow \mathtt{S}\langle \mathtt{N}(n) \rangle \quad \text{[N-S-i]}$$

$$\langle n \rangle \vdash \mathtt{E\_S} \Rightarrow \mathtt{S}\langle \mathtt{E}(n) \rangle \quad \text{[E-S-i]}$$

$$\langle n \rangle \vdash \mathtt{N\_Q} \Rightarrow \mathtt{Q}\langle \mathtt{N}(n) \rangle \quad \text{[N-Q-i]}$$

$$\langle n \rangle \vdash \mathtt{E\_Q} \Rightarrow \mathtt{Q}\langle \mathtt{E}(n) \rangle \quad \text{[E-Q-i]}$$

$$\langle n \rangle \vdash \mathtt{N\_O} \Rightarrow \mathtt{O}\langle \mathtt{N}(n) \rangle \quad \text{[N-O-i]}$$

$$\langle n \rangle \vdash \mathtt{E\_O} \Rightarrow \mathtt{O}\langle \mathtt{E}(n) \rangle \quad \text{[E-O-i]}$$

$$\langle \ldots, n, \ldots \rangle \vdash \mathtt{N\_S}(n) \Rightarrow \mathtt{S}\langle \mathtt{N}(n) \rangle \quad \text{[N-S]}$$

$$\langle \ldots, n, \ldots \rangle \vdash \mathtt{E\_S}(n) \Rightarrow \mathtt{S}\langle \mathtt{E}(n) \rangle \quad \text{[E-S]}$$

$$\langle \ldots, n, \ldots \rangle \vdash \mathtt{N\_Q}(n) \Rightarrow \mathtt{Q}\langle \mathtt{N}(n) \rangle \quad \text{[N-Q]}$$

$$\langle \ldots, n, \ldots \rangle \vdash \mathtt{E\_Q}(n) \Rightarrow \mathtt{Q}\langle \mathtt{E}(n) \rangle \quad \text{[E-Q]}$$

$$\langle \ldots, n, \ldots \rangle \vdash \mathtt{N\_O}(n) \Rightarrow \mathtt{O}\langle \mathtt{N}(n) \rangle \quad \text{[N-O]}$$

$$\langle \ldots, n, \ldots \rangle \vdash \mathtt{E\_O}(n) \Rightarrow \mathtt{O}\langle \mathtt{E}(n) \rangle \quad \text{[E-O]}$$

$$\frac{\gamma \vdash t_c \Rightarrow \tau_c}{\gamma \vdash \mathtt{collection}\langle t_c \rangle \Rightarrow \mathtt{Q}\langle \tau_c \rangle} \quad \text{[CC]}$$

$$\frac{\gamma = \langle \ldots, n, \ldots \rangle \ \wedge \ \gamma \vdash t_{pc} \Rightarrow \tau_{pc}}{\gamma \vdash \mathtt{N\_P}\langle t_{pc} \rangle(n) \Rightarrow \mathtt{P}\langle \mathtt{N}(n), \tau_{pc} \rangle} \quad \text{[n-p]}$$

$$\frac{\gamma = \langle n \rangle \ \wedge \ \gamma \vdash t_{pc} \Rightarrow \tau_{pc}}{\gamma \vdash \mathtt{N\_P}\langle t_{pc} \rangle \Rightarrow \mathtt{P}\langle \mathtt{N}(n), \tau_{pc} \rangle} \quad \text{[n-p-i]}$$

$$\frac{\gamma = \langle \ldots, n, \ldots \rangle \ \wedge \ \gamma \vdash t_{pc} \Rightarrow \tau_{pc}}{\gamma \vdash \mathtt{E\_P}\langle t_{pc} \rangle(n) \Rightarrow \mathtt{P}\langle \mathtt{E}(n), \tau_{pc} \rangle} \quad \text{[e-p]}$$

$$\frac{\gamma = \langle n \rangle \ \wedge \ \gamma \vdash t_{pc} \Rightarrow \tau_{pc}}{\gamma \vdash \mathtt{E\_P}\langle t_{pc} \rangle \Rightarrow \mathtt{P}\langle \mathtt{E}(n), \tau_{pc} \rangle} \quad \text{[e-p-i]}$$

$$\frac{\gamma \vdash t_{kv} \Rightarrow \tau_{kv} \ \wedge \ \tau_{kv} \neq \mathtt{bool} \ \wedge \ \gamma \vdash t'_{kv} \Rightarrow \tau'_{kv} \ \wedge \ \tau'_{kv} \neq \mathtt{bool}}{\gamma \vdash \mathtt{map}\langle t_{kv}, t'_{kv} \rangle \Rightarrow \mathtt{map}\langle \tau_{kv}, \tau'_{kv} \rangle} \quad \text{[m]}$$

Figure A.4 – GREEN-MARL type translations.