COMPILER FRONT END FOR GUARDOL – A DOMAIN-SPECIFIC LANGUAGE FOR HIGH ASSURANCE GUARDS

by

JONATHAN COLE HOAG

B.S., Kansas State University, 2007

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2010

Approved by:

Major Professor
Dr. John Hatcliff

Abstract

Guardol, a domain-specific language (DSL) developed by Rockwell Collins, was designed to streamline the process of specifying, implementing, and verifying Cross Domain Solution (CDS) security policies. Guardol's syntax and intended computational behavior tightly resembles the core of many functional programming languages, but a number of features have been added to ease the development of high assurance cross domain solutions. A significant portion of the formalization and implementation of Guardol's grammar and type system was performed by the SAnToS group at Kansas State University.

This report summarizes the key conceptual components of Guardol's grammar and toolchain architecture. The focus of the report is a detailed description of Guardol's type system implementation and formalization. A great deal of effort was put into a formalization which provided a high level of assurance that the specification of types and data structures were maintained in the intended implementation.

Table of Contents

Table of Contents iv

List of Figures vii

Acknowledgements viii

1 Introduction 1
1.1 Background . . . 1
1.2 Cross Domain Solutions . . . 2
1.2.1 Existing Guards . . . 3
1.3 Guardol Motivation . . . 3
1.3.1 Goals . . . 4
1.4 Contributions of this Report . . . 5

2 Guardol By Example 8
2.1 High-level policy . . . 8
2.2 Type declarations . . . 9
2.2.1 Qualified Type Names . . . 9
2.2.2 List Types . . . 13
2.2.3 Union Types . . . 13
2.2.4 Record Types . . . 13
2.2.5 Combining Composite Types . . . 14
2.3 Node declarations . . . 14

3 Guardol Compiler/Toolchain Architecture 18
3.1 Parser . . . 18
3.2 Abstract Syntax Tree . . . 21
3.3 XML Builder . . . 22
3.4 IR Translation . . . 23
3.5 KSU Analysis and Verification . . . 25
3.6 Lustre Back-end . . . 25

4 Types 26
4.1 Motivation . . . 26
4.1.1 Abstraction . . . 27
4.1.2 Safety . . . 27
4.1.3 Error Detection . . . 28
4.1.4 Documentation . . . 28

4.2 Primitive Types 28
4.3 User Defined Types 29
4.3.1 Simple Types 30
4.3.2 List Types 31
4.3.3 Tuple Types 32
4.3.4 Record Types 33
4.3.5 Union Types 34
4.4 Recursive and Mutually Recursive Types 36
4.5 Type Equivalence 36
4.5.1 Overview 36
4.5.2 Motivation 37
4.5.3 Guardol Type Equivalence 38
4.5.4 Type Environment 40
4.5.5 Example Derivations 43
4.5.6 Assessment 44

5 Type Constraints **45**
5.1 Syntax Domain 45
5.2 Data Structures 46
5.3 Definedness Extension of Type Equivalence 49
5.4 Rule Form 50
5.5 Type Constraint Rules 51
5.6 Union-Find 55
5.7 Type Constraint Generation Example 59
5.7.1 Step 1 60
5.7.2 Step 2 61
5.7.3 Step 3 61
5.7.4 Step 4 63
5.7.5 Step 5 63
5.7.6 Step 6 64
5.7.7 Step 7 64

6 Future Work **67**
6.1 Specification Language 67
6.2 Pilar Intermediate Representation 68
6.3 Verification 69

Bibliography **70**

A Grammar **71**
A.1 Names 71
A.2 Type Declarations 72
A.3 Constant Declarations 74

A.4 Expressions 74
A.5 Statements 78
A.6 Node Declarations 79
A.7 Package Declarations 80

B ANTLR Grammar **82**

C Guardol AST Model **94**

List of Figures

2.1 Sample GIL code for an MIME email guard (part 1) 10
2.2 Sample GIL code for an MIME email guard (part 2) 11
2.3 Sample GIL code for an MIME email guard (part 3) 12

3.1 Guardol Compiler/Toolchain Architecture 19
3.2 Excerpts from “model.plr” 21
3.3 Example XML output of Guardol program 23
3.4 A fragment of Guardol source code 24
3.5 Pilar intermediate representation 24

4.1 Sample GIL code for demonstrating type constraints 30
4.2 Formal Definition of Type Equivalence 41
4.3 Formal Definition of the Type Structure Lookup Operation 42

5.1 Type Constraint Syntax Domain 46
5.2 Definedness Extension of Type Equivalence 49
5.3 AST - Step 1 61
5.4 Cache ($\hat{C}$) - Step 1 61
5.5 AST - Step 2 62
5.6 Cache ($\hat{C}$) - Step 2 62
5.7 AST - Step 3 62
5.8 Cache ($\hat{C}$) - Step 3 62
5.9 AST - Step 4 63
5.10 Cache ($\hat{C}$) - Step 4 63
5.11 AST - Step 5 64
5.12 Cache ($\hat{C}$) - Step 5 64
5.13 AST - Step 6 65
5.14 Cache ($\hat{C}$) - Step 6 65
5.15 AST - Step 7 66
5.16 Cache ($\hat{C}$) - Step 7 66

Acknowledgments

First and foremost, many thanks to Dr. John Hatcliff. I imagine there are very few others whose patience and expert guidance could have seen this project to completion. I am grateful also to the rest of my committee: Dr. Torben Amtoft, Dr. Robby, and Dr. David Schmidt for their wealth of knowledge and tolerance of constant inquiry. The opportunity provided by Rockwell Collins and the chance to work with everyone from the Cedar Rapids office was one I will never forget. Countless tele-conferences and the occasional visit were an invaluable resource and this project would simply not exist without them.

Thanks also to my parents, Richard and Linda Hoag, who always provided a silent motivation to never give up; not to discount their enthusiastic encouragement despite my many tangents and often incoherent ramblings in pursuit of a concise justification of the importance of this project.

Chapter 1

Introduction

1.1 Background

Modern information systems are highly networked and distributed systems enabling the transfer of information across multiple platforms and disseminating it using a diverse set of networking policies. The information system, which we will refer to as the composition of these platforms and networking policies, provides the infrastructure for communication between the different domains. The network policies abstract over much of the platform specific implementation and specify properties about the information passing through the system. This report focuses on the specification of security policies, from the simple governmental security classifications such as TOP SECRET or RESTRICTED information and their allowable destinations to more complex policies.

A Cross Domain Solution (CDS) is “an information assurance solution that provides the ability to manually and/or automatically access and/or transfer data between two or more differing security domains” [1]. For security critical systems, each network or platform may have its own security domain and set of security policies that govern the flow of data through the system for which a CDS must be designed. In the case of an information system that utilizes security classifications, it is understood that each classification may have its own security domain and require that information be transferred between the secure networks of different classifications. However, it can be assumed that not all information should be able to pass between networks with different classifications. Thus, we have need of a security policy that determines what information is safe to transfer between corresponding security domains and what information must be altered or discarded. In this scenario, the security policy would be specified in the CDS and the information system would be implemented such that all information passing between the networks goes through the CDS and is subject to its security policy.

1.2 Cross Domain Solutions

Information passing through differing security domains provides unique information security obstacles with the potential for sensitive information to "leak" to unauthorized users. A CDS is built when information security is of the utmost importance to one or more of these domains. Cross domain information sharing, often thought of as a necessary evil, is found more frequently in industry and government due to the shear quantity of information required by specific domains as well as the increased need for the immediate dispersion of urgent intelligence despite its classification and origin.

Abstractly, a CDS should be thought of as a high-level security policy that is automatically enforced by a message passing system. This abstraction allows the technical specification of the CDS security policy to be easily understood. For instance, a written requirement that a message marked as TOP SECRET can only be sent to the United Kingdom when its destination is London seems straightforward. However even for simple specifications such as this, it is often difficult to verify that the CDS implements the intended security policy. These problems are best expressed by The Anderson Panel Report of 1972 where it is understood that "a large part of the design problem is attributable to the absence of models as a medium for translating security requirements to technical specification and as a source of acceptance criteria for evaluating the product" [2].

1.2.1 Existing Guards

A number of potential Cross Domain Solutions exist for use in industry today. Many, however, are targeted for specific end-to-end domains. For example, "General Dynamics' Trusted Network Environment (TNE) has a flexible range of Commercial off-the-shelf (COTS) hardware and software that can be customized to fit the network"[3]. The TNE is dependent on program requirements that must be customized by a General Dynamics team for each client. While this does have the potential to provide a secure environment based on a specific set of security policies, the environment is restricted to the domain built by the General Dynamics team. These domain restrictions, e.g. forcing the domains to be same, imply that this solution is more of a single domain solution than a CDS.

Problems with the existing CDS solutions stem from the need for trained technicians to administer guard policies. The cost of these solutions is difficult to predict with modular or frequently changing security policies because the static security policies must constantly be updated. These solutions are also unadaptable to quickly changing policies needed for rapid response because they must be manually configured. Limitations caused by platform dependencies are also a hazard to the distribution of these solutions.

1.3 Guardol Motivation

The Guardol development effort, sponsored by the United States Department of Defense, was undertaken to create a domain-specific language (DSL) that would enable the specification, implementation, verification, and standardization of high assurance guard solutions. Guards, as they will be referred to in this document, provide the connections used in within the CDS. The implementations of these guards provide an automated mechanism for communication between specific security domains where communications are restricted and enforced by the functionality of the guard.

Currently, the development and maintenance of high assurance guards is an extremely expensive endeavor. Most guards are targeted for use on platforms dedicated to the guarding application where the guard is expensive to buy and expensive to configure. Since guards are often developed utilizing static security policies, they are hugely cumbersome and costly to change. The absence of a standardized framework for potential platforms also restricts the ability of guarding specifications and implementations to be quickly adapted to changing risks and alternate platforms.

In an effort to surmount the disadvantages of developing fixed guards for dedicated platforms, Rockwell Collins Advanced Technology Center is developing a DSL, developed to specifically target the creation of high assurance guards. The SAnToS group at Kansas State University has received a contract from Rockwell Collins to assist in the design of the Guardol Language and implement a prototype compiler. It is hoped that this DSL will provide a means to design flexible guarding policies, a mechanism by which guarding policies can be specified separately from their actual implementations and execution platforms, and a high assurance that these guards function correctly. Also, with the expanded use of a DSL for guard specific applications, it is anticipated that a standardization of guarding policies will emerge that facilitates faster development and higher assurance of correct functionality.

1.3.1 Goals

The following themes guide the design of the Guardol DSL.

1. *Domain Appropriateness* – The Guardol DSL should not aim to be a general purpose programming language. Instead, it should focus on providing built-in mechanisms that facilitate rapid development and verification of guarding functionality. This implies that it should focus primarily on providing language constructs that facilitate descriptions of packets of data, deconstruction of packets into individual fields, operations to examine and transform form packet fields, and reconstruction of packets from fields. The language should omit features to support numeric computation, dynamic creation of storage, etc. non-crucial to the guarding domain. In situations where such functionality is needed by a guard application, it will be achieved via external programs

that are made visible to the application as external functions.

2. *Extensibility* – Recognizing that guards will be applied in a variety of contexts that cannot be anticipated presently, the DSL should support addition of new types of data, new predicates and formal claims to be used in specification and verification, new forms of data checkers and transformers (both internal and external). The DSL should provide mechanisms for organizing these extensions into libraries of usable artifacts.

3. *Analyzability* – Due to the high-assurance nature of the domain, the DSL should be designed to facilitate highly automated analysis for both functional correctness and information flow properties, and analysis to determine if guard design corresponds to implementations that stay within the resource bounds of a target platform. The DSL and associated analysis components should be designed to ensure that : (a) analysis can take place as early in the life cycle as possible, and (b) analysis can be applied and interpreted by developers that may not be experts in formal methods.

4. *Platform Independence and Retargetability* – The DSL should support rendering of guard descriptions in a manner that allows an implementation to be generated in either hardware or software and in a manner that allows a guard originally implemented on one platform to be subsequently implemented on a different platform.

1.4 Contributions of this Report

This report describes the implementation of a compiler for Guardol including much of the language's base architecture and its corresponding tool chain. This MS project, however, encompasses a large development effort beyond the design of the language itself. The language proper consists of the Guardol grammar, parser, Abstract Syntax Tree (AST) representation, Pilar Intermediate Representation (Pilar IR), and type system. This project provides a foundation upon which a large number of verification analysis, external tools, target platform translations, and a property language can be implemented to further the flexibility of the final product.

The contributions of this MS project are:

- a justification for the Guardol DSL development project.
- a demonstration of the use of the Guardol core language.
- a description and specification of the current Guardol language architecture and implementation.
- an XML representation of the Guardol AST.
- a translation to the Pilar IR such that existing static analyses can be leveraged.
- a type system that leverages the correlation between name-centric type specification and high assurance guard domains.
- a formal definition of the Guardol language's type equivalence and type constraint rules such that Guardol's type system can be rigorously evaluated.
- a framework to reason about and later incorporate verification analysis, external tools, target platform translations, and the creation of a property language.

The grammar of a language is a fundamental formalization that must be finalized before much of the language development can begin. Presently, the Guardol grammar has been finalized in two different forms: the formal grammar exhibited in Appendix A and the adapted ANTLR grammar, Appendix B, created to perform parsing and the construction of the Guardol AST. Guardol, however, is an experimental language, so it is conceivable that further changes to the grammar are possible. Any syntax changes or extension that cause a significant modification to the grammar will invariably require adjustments to the entire existing tool chain. At present, no changes to the grammar are expected.

The Guardol parser is constructed using the ANTLR parser generator (http://antlr.org/). ANTLR is an open source parser generator that processes LL* grammars[4], which allows grammar productions to be written more naturally (*e.g.*, without worry about shift/reduce conflicts). Parsing errors produce messages that can be displayed in, for example, console output. Then the Guardol parser constructs an AST using classes custom designed model generating scripts. The modeling file used to construct the AST classes is exhibited in Appendix C.

The Pilar IR and XML-based representation have also been finalized for the Guardol profile. Currently, these two translations are implemented through a traversal of the AST. The XML-based representation is used exclusively for an external audit by Rockwell Collins proprietary software. The Pilar IR provides a foundation for the use of verification analysis and translations to target platforms that have not yet been implemented.

The type system is built using the Pilar IR. A traversal of the Pilar IR, similar to that of the AST, is done after the translation to the Pilar IR is completed. This traversal encodes the type equivalence and type constraint generation rules demonstrated in Chapter 4 and Chapter 5. Type checking errors are output to the console should the type constraint generation phase fail.

Chapter 2

Guardol By Example

In this section, we illustrate the key language features of the Guardol Implementation Language, GIL, with a simple example. The example is a guard application which takes an input email message in MIME format and checks/transforms the message based on a policy. The example is reasonably challenging because it involves recursive message structure, multiple fields with varying types occurring as subcomponents to the message, and a variety of checks/transformations for each type.

2.1 High-level policy

Below we list policies that the guard application should enforce.

1. The MIME message must go through a virus check and if virus is found the message is blocked.

2. The body of the email shall go through a "dirty-word search" process, in which words in the message body are looked up in a dictionary of disallowed words. If a "dirty word" is found the message is blocked.

3. If the message contains an attachment whose name has an executable-file suffix (.exe, .com, .dll, .o, .vba, .zip), the message is blocked.

4. If an attachment is a MIME message it is subject to the same check as the guard performs.

5. If an attachment is a textual file, the file is subject to dirty-word search.

6. If the attachment is an XML file, it is subject to a check called "XML_DOM_CHECK".

7. If the attachment is a binary file, it will be checked to see whether the binary file is an executable (even if the file's name does not have an executable-file suffix). If it is indeed an executable the message is blocked.

8. All other types of attachments are blocked.

9. Whenever a MIME message is rejected by the guard, an audit log shall record the reason why it is blocked.

The complete code of the MIME guard in GIL is illustrated in Figure 2.1 - 2.3. Below we explain the key parts of the language by walking through this simple example.

2.2 Type declarations

The general form of type declaration in GIL is **type** ID **is** Type, where ID is the name of the type being declared, and Type is the definition of the type.

2.2.1 Qualified Type Names

The following declaration illustrates how the current package uses (and can potentially rename) a type declared in another GIL package (the definition of the GIL package system is not finalized and we do not discuss it further in this document).

```
type XML_DOM_Type is package1.XML_DOM_Type;
```

```
package MIME is

type XML_DOM_Type;

type stringList is list string end list;

type ByteList is list byte end list;

type AttachmentType is union
    MIME_Email of MIME_Type |
    Text of string |
    XML_DOM of XML_DOM_Type |
    Binary of ByteList |
    Other
end union;

type Attachment is record
   name : string,
   object : AttachmentType
end record;

type AttachmentList is list Attachment end list;

type MIME_Type is record
    body : string,
    attachments : AttachmentList
end record;

type  AuditMsg is union
      Dirty_Word_Check_Failed |
      Virus_Check_Failed |
      Exe_Suffix_Check_Failed |
      Exec_Check_Failed |
      Unknown_Object_Type
end union;

node VIRUS_CHECK
   (Input : MIME_Type) returns
   (ok : bool)
is external;

node DIRTY_WORD_SEARCH
   (text : string) returns
   (newText : string, Audit : AuditMsg)
is external;

node EXEC_CHECK
   (input : ByteList) returns
   (ok : bool)
```

Figure 2.1: Sample GIL code for an MIME email guard (part 1)

```
is external;

node NOT_SUFFIX
  (input : string, suffixes: stringList) returns
  (ok : bool)
is external;

node XML_DOM_CHECK
  (input : XML_DOM_Type) returns
  (newXML : XML_DOM_Type, Audit : AuditMsg)
is external;

node MIME_Check
 (Input : MIME_Type) returns
 (Output : MIME_Type, Audit : AuditMsg)
is
local
  virusOk : bool;
  newAttachments : AttachmentList;
  newBody : string;
  attachAudit : AuditMsg;
  dirtyAudit : AuditMsg;
begin
  virusOk := VIRUS_CHECK(Input);
  newBody,dirtyAudit := DIRTY_WORD_SEARCH(Input.body) when virusOk;
  attachAudit, newAttachments := attachmentListCheck(Input.attachments);
  Output := MIME_Type'[body => newBody, attachments => newAttachments];
  Audit := attachAudit default
    (AuditMsg'Dirty_Word_Check_Failed when (not exists newBody)) default
      (AuditMsg'Virus_Check_Failed when (not virusOk));
end node;

node attachmentListCheck
  (Input : AttachmentList) returns
  (audit : AuditMsg, newList : AttachmentList)
is
local
  newHd : Attachment;
  newTl : AttachmentList;
  auditHd : AuditMsg;
  auditTl : AuditMsg;
  hd : Attachment;
  tl : AttachmentList;
begin
  match Input with
    hd::tl =>
      auditHd, newHd := attachmentCheck(hd);
      auditTl, newTl := attachmentListCheck(tl);
      newList := (newHd::newTl);
```

Figure 2.2: Sample GIL code for an MIME email guard (part 2)

```
      audit := auditHd default auditTl;
  | _ => skip;
  end match;
end node;

node attachmentCheck
  (Input : Attachment) returns
  (Audit : AuditMsg, newAttachment : Attachment)
is
local
  isNotExeSuffix : bool;
  newObject : AttachmentType;
  objectAudit : AuditMsg;
  newMail : MIME_Type;
  newText : string;
  newElem : XML_DOM_Type;
  wellFormed : bool;
  mail : MIME_Type;
  text : string;
  elem : XML_DOM_Type;
  bin : ByteList;
begin
  isNotExeSuffix :=
    NOT_SUFFIX(Input.name,
               stringList'{".exe", ".com", ".dll",
                           ".o", ".vba", ".zip"});
  match Input.object with
    MIME_Email mail =>
      newMail,objectAudit := MIME_Check(mail);
      newObject := AttachmentType'MIME_Email(newMail);
  | Text text =>
      newText, objectAudit := DIRTY_WORD_SEARCH(text);
      newObject := AttachmentType'Text(newText);
  | XML_DOM elem =>
      newElem, objectAudit := XML_DOM_CHECK(elem);
      newObject := AttachmentType'XML_DOM(newElem);
  | Binary bin =>
      wellFormed := EXEC_CHECK(bin);
      newObject := Input.object when wellFormed;
      objectAudit := AuditMsg'Exec_Check_Failed when (not wellFormed);
  | Other =>
      newObject := Input.object when false;
      objectAudit := AuditMsg'Unknown_Object_Type;
  end match;
  Audit := (AuditMsg'Exe_Suffix_Check_Failed when (not isNotExeSuffix))
             default objectAudit;
  newAttachment := Attachment'[name => Input.name, object => newObject];
end node;
end package;
```

Figure 2.3: Sample GIL code for an MIME email guard (part 3)

2.2.2 List Types

GIL supports parameterized (generic) types such as StringList. However, the language requires that all types be named. The following type declarations give specific names (StringList and ByteList) to instantiations of the generic type String.

```
type StringList is String list;
```

```
type ByteList is byte list;
```

2.2.3 Union Types

A union type provides the capability of expressing choices for various possible data shapes. It consists of multiple branches corresponding to the multiple choices. Each branch is given a name and the type of the data when the branch is chosen, in the form of Name **of** Type. For example, the type of an attachment's data content could be one of MIME, Text, XML, Binary, or others. This can be expressed in the following union type declaration.

```
type AttachmentType is union
    MIME_Email of MIME_Type |
    Text of String |
    XML_DOM of XML_DOM_Type |
    Binary of ByteList |
    Other
end union;
```

2.2.4 Record Types

A record type specifies a composite data structure that consists of one or more fields. For example, an attachment contains a field that specifies the name of the attachment and the actual data content. Thus the type Attachment can be expressed as a record type as below. The object field is the data content and its type is the AttachmentType specified above.

```
type Attachment is record
  name : String,
  object : attachmentType
end record;
```

We give a named type for a list of attachments.

```
type AttachmentList is Attachment list;
```

An MIME message has a message body which is a string of characters, and zero or more attachments. This can be expressed in the following record type.

```
type MIME_Type is record
   body : String,
   attachments : AttachmentList
end record;
```

2.2.5 Combining Composite Types

We also define a type for audit messages, which is a union type with each branch indicating a reason for the message to be blocked. In this example, we consider only a very simple notion of audit messages – audit messages are just tags indicating the kind of guarding failure that occurred. Thus, in this union type definition the branches do not have a corresponding type which means there is no data associated with each branch. The name of the branch is the only information a value of the type conveys.

```
type  AuditMsg is union
     Dirty_Word_Check_Failed |
     Virus_Check_Failed |
     Exe_Suffix_Check_Failed |
     Exec_Check_Failed
end union;
```

2.3 Node declarations

The functionality of a guard is expressed in the units of "nodes" in GIL. A node is a subroutine that takes zero or more inputs and produces zero or more outputs. The general form of node definition in GIL is

```
node NodeName
     InputParameters
     returns
     OutputParameters
is
local
     LocalParameters
begin
     Statements
end node;
```

The "local" block is optional and is used to declare local variables in addition to the input and output parameters. Node bodies cannot reference non-local variables. The following node defines the top-level function of checking an MIME message.

```
node MIME_Check
 (Input : MIME_Type) returns
 (Output : MIME_Type, Audit : AuditMsg)
is

local
  virusOk : Boolean;
  newAttachments : AttachmentList;
  newBody : String;
  attachAudit : AuditMsg;
  dirtyAudit : AuditMsg;

begin
  virusOk := VIRUS_CHECK(Input);
  newBody, dirtyAudit := DIRTY_WORD_SEARCH(Input.body) when virusOk;
  newAttachments, attachAudit := attachmentListCheck(Input.attachments);
  Output := MIME_Type'[body => newBody, attachments => newAttachments];
  Audit := attachAudit default
    (AuditMsg'Dirty_Word_Check_Failed when (not exists newBody)) default
      (AuditMsg'Virus_Check_Failed when (not virusOk));
end node;
```

The node takes an input parameter of type MIME_Type, and returns two values which are the transformed message (Output) if the message is allowed, and an audit message (AuditMsg) if the message is blocked.

The node body is composed of a number of statements corresponding to the various checks performed on the message. Each of the statements is an assignment of an expression value to a variable. In GIL, a variable can be assigned at most once. The value of an unassigned variable is undefined. GIL supports a variety of expressions some of which could also produce an undefined value. As one example of such expressions, if a function's node body does not assign a value for an output parameter, the expression that applies the function will return an undefined value for that output. If such an expression happens to be the right hand side of an assignment statement, the left hand side variable will still be undefined.

At line number 14, the virus check is performed by invoking the function VIRUS_CHECK on the input message. The function returns a Boolean value which is assigned to a local variable virusOk. At line number 15, the dirty-word search is performed on the message body *only when the virus check has succeeded.* The "when" expression will evaluate to the left operand if the right operand evaluates to the "true" Boolean value. If the right operand evaluates to "false" or is undefined, the whole "when" expression is undefined. This means that local variable newBody and dirtyAudit will be defined *only if* virusOk is "true". At line number 16, the attachments are checked by invoking the attachmentListCheck function on the attachments field of the input message. The check returns a transformed attachment lists assigned to newAttachments, and an audit message assigned to attachaudit.

At this point, we know that both newBody and newAttachments could be undefined, and if the message should be blocked due to problems in its body or attachment, we shall find the reason in virusOk, dirtyAudit, and attachAudit. So we can construct the new transformed message and the audit message. To construct a value with a type T, GIL uses the expression T'DataContent. For the case of record type, the corresponding fields are given values in DataContent. For the case of union type, the chosen branch and associated data value are given in DataContent, and so on. The newly constructed MIME message will have newBody as its body, and newAttachments as its attachments. An interesting question is what happens when one (or both) of them is undefined? GIL supports a "strict" evaluation semantics in which case if a component of an expression is undefined, the whole expression becomes undefined. Under the strict semantics, the guard will discard the whole message when a component of it violates the policy. Alternatively, if we adopt a non-strict semantics, the violating components will be stripped off the message and the remainder of the message will still be allowed. Another interesting question is when more than one component in a message violates the policy, what audit message shall the guard produce? The guard uses a "default" expression to provide a precedence in outputting audit messages. Informally, the expression P **default** Q will evaluate to P if P is defined, otherwise it will evaluate to Q. By chaining a number of “default” expressions we can define the precedence of outputting audit messages. In this example, the audit from checking attachments has the highest priority. If there is no audit from the attachment, then the audit of dirty word search is used. The expression uses **when not exists** newBody as the condition to indicate dirty-word search has failed. **exists** V returns a Boolean value that is “true” when V is defined. If dirty-word search does not find any problem either, then the result of virus check is used to construct the audit message.

The other functions in the guard are defined in a similar manner. Some functions are actually interfaces for external capabilities such virus checking and data scrubbing. In these cases the node declaration explicitly uses the **external** directive and no definition of the body needs to be given. For example:

```
node VIRUS_CHECK
  (Input : MIME_Type) returns
  (ok : Boolean)
is external;

node DIRTY_WORD_SEARCH
  (text : String) returns
  (newText : String, Audit : AuditMsg)
is external;
```

Chapter 3

Guardol Compiler/Toolchain Architecture

Figure 3.1 presents the architecture of the Guardol compiler built during this MS project and its relationship to other tool capabilities being developed in the Guardol project. The compiler and supporting analysis are developed using Kansas State's Sireum (`http://sireum.org`) analysis and verification framework and a number of widely used open source packages such as the ANTLR parser generator and XStream XML manipulation libraries. Sireum itself is publicly available under the open source Eclipse Public License. The implementation specific to this MS project is comprised of over 20000 lines of Java code, where 15000 were auto-generated by tools. About 6000 lines of code were written to facilitate the IR translation, type checking algorithms, as well as the scripting code used to auto-generate Java code for the parser and AST. Below we provide an overview of each component of the compiler architecture.

3.1 Parser

The Guardol parser is constructed using the ANTLR parser generator (`http://antlr.org/`). ANTLR is an open source parser generator that processes LL* grammars[4], which allows grammar productions to be written more naturally (*e.g.*, without worry about shift/reduce conflicts). Below is an example production rule written in ANTLR's grammar lan-

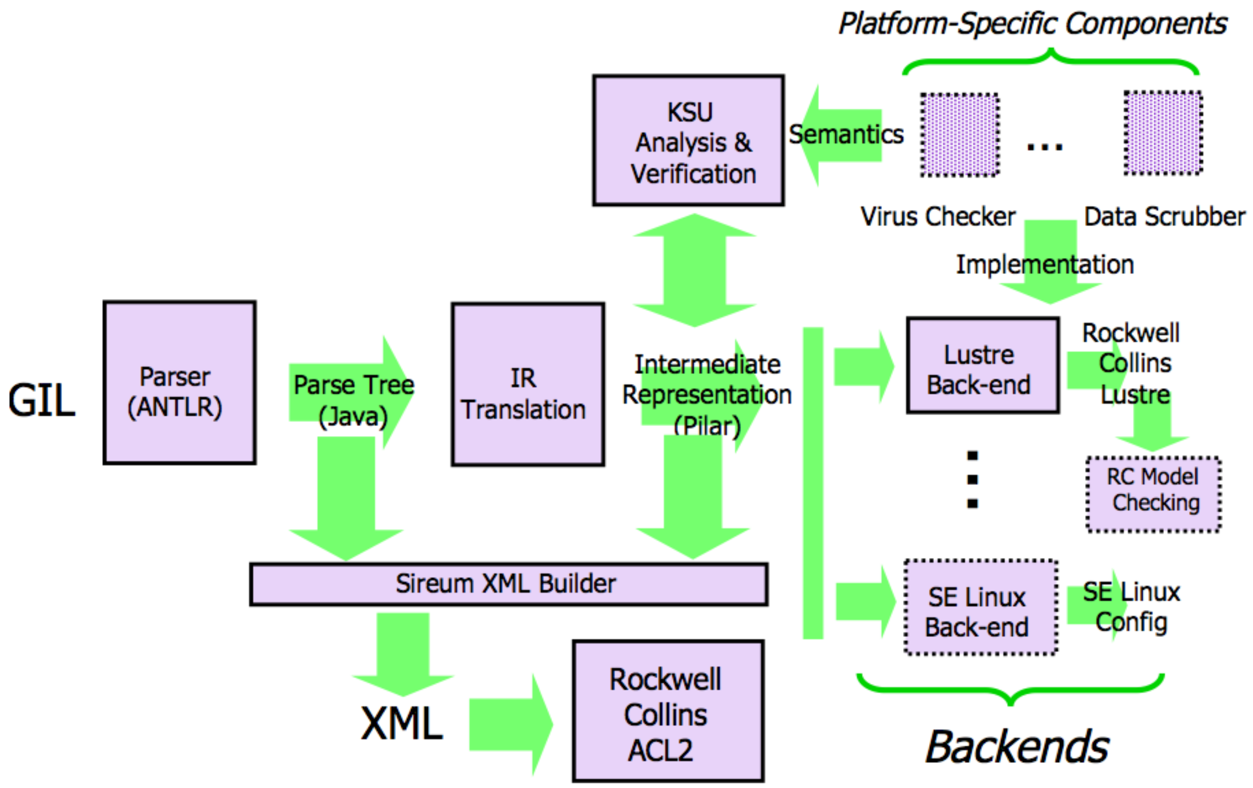

Figure 3.1: Guardol Compiler/Toolchain Architecture

guage.

```
assignmentStatement returns
[ AssignmentStatement result = new AssignmentStatement() ]
@init {ArrayList<Name> names = new ArrayList<Name>();}
:
    (n=nameIdentifier  {names.add($n.result);}
      (',' n=nameIdentifier {names.add($n.result);}
       )*
     )

   ':=' e=expression l=';'{result.setTheExp($e.result);}
    {result.setTheNames(names);}
    {result.setTheSelection(
      new RegionSelection(
        names.get(0).getTheSelection().getStart(),
        new Caret($l.line,$l.pos),false));}
;
```

The ANTLR production rule above specifies the rule for parsing and generating the

Abstract Syntax Tree (AST) for an assignment statement, i.e., the ANTLR production rule that corresponds to the Guardol grammar rule for *assignmentStatement* (A.63). The BNF production rule can be lifted directly from the text when you exclude text that is surrounded by brackets or curly braces and other keywords used to construct the Java AST. The distilled BNF rule in ANTLR is

```
assignmentStatement :
  nameIdentifier (',' nameIdentifier)* ':=' expression ';'
;
```

where *nameIdentifier* and *expression* other production rules within the ANTLR grammar. This BNF rule requires the left hand side of an assignment can be constructed with one name or multiple names separated by a comma and the right hand side of the assignment statement be an expression.

The code wrapped in curly braces and brackets is used to help ANTLR automatically generate a custom AST from the grammar. For instance, the [AssignmentStatement result = new AssignmentStatement()] code tells ANTLR to create a new AssignmentStatement object every time the *assignmentStatement* production rule is used. The AssignmentStatement object's class is part of an automatically generated set of Java classes representing the Java AST produced by the Sireum class design module.

ANTLR is widely used in both academia and industry (e.g. Oracle and Microsoft). It produces a parse tree represented as a Java data structure, and provides good support for tree construction, tree walking, translation, error recovery, and error reporting. The parse tree that results the compiler parsing phase can either be translated to the Pilar IR (used in analysis/verification and further translation to target platforms or languages, e.g. Lustre), or it can be used to provide an XML-based external representation that can be utilized by other tools such as the Rockwell Collins ACL2 verification tool chain. The ANTLR grammar definition file produced for this MS Project can be found in Appendix B.

```
//The generic AST Node
//Contains:
//-A selection object referring to the starting/ending line numbers/char offsets
record abstract Node
{
        org::sireum::profile::guardol::selection::IRegionSelection theSelection
                @Default new org.sireum.profile.guardol.selection.RegionSelection();
}

//Statement
//--Abstract definition of a statement
record abstract Statement extends Node
{

}

//Assignment Statement
//Contains:
//-A list of names identifying the assignment of the exp to these variables
//-An exp that is used for the assignment
record AssignmentStatement extends Statement
{
        Name[] theNames
                @Default ^[];
        Exp theExp;
}
```

Figure 3.2: Excerpts from “model.plr”

3.2 Abstract Syntax Tree

Although ANTLR does provide a mechanism for the implicit construction of a Concrete Syntax Tree (CST) based on the production rules enumerated in the Guardol grammar, it was too unwieldy to manage the translation to the Pilar IR and the XML-based external representation. Instead, the Guardol parser generates the AST using modifications to the ANTLR grammar as described in Section 3.1. The Guardol Abstract Syntax Tree (AST) is constructed using a collection of Java classes generated by a Sireum class design module. The file used to generate the AST in Appendix C briefly summarizes the purpose of the AST nodes. For example, the last record in Figure 3.2 models the creation of the AssignmentStatement class. AssignmentStatement is an extension of the abstract Statement class such that it inherits all of the fields provided by the abstract Statement and abstract Node class. The Node class represents the generic AST node containing fields for the storage of line number and char offset markers specifying the AST node’s position in the parsed text.

The AssignmentStatement class specifies two fields corresponding to the AST nodes that are contained within an *assignmentStatement*

- A list of name AST nodes – the variable names to be assigned
- An expression AST node – the expression that will be assigned

All classes created in this way will extend the generic AST Node class such that line number and char offset information can be gathered for every node in the AST.

The Sireum class design module automatically generates an AST node traversal class known as a Visitor. This Visitor class is the product of the well known visitor design pattern that "separates an algorithm from an object structure by moving the hierarchy of methods into one object"[5]. Using a simple overloading of the Visitor's empty visit methods representing each AST node, the Visitor can be used to translate the AST into the Pilar IR and the XML-based external representation. At this point, only the rules enforced by the grammar cause any error messages to be generated. Type checking and other verification analyses are performed after the AST has been translated into the Pilar IR.

3.3 XML Builder

The XML Builder provides a convenient mechanism for generating an XML-based external representation of a parse tree using the XStream framework (http://xstream.codehaus.org/).

XStream provides API to serialize any Java object to XML and also to deserialize it back without any schema definitions. Class structures are used as the XML schema by default, while allowing custom serializers/deserializers to be specified for flexibility. XStream has been used in a lot of commercial and open source projects in production code such as Atlassian Confluence (http://www.atlassian.com/software/confluence/) and Apache Muse (http://ws.apache.org/muse/), and it is supported by a variety of frameworks such as jBoss ESB (http://www.jboss.org/jbossesb/) and Mule (http://www.mulesource.org/).

```
<AssignmentStatement>
   <NameList><IDName><ID><String>
      virusOK
   </String></ID></IDName></NameList>
   <CallExp><ID><String></String></ID>
     <ID><String>
       VIRUS_CHECK
     </String></ID>
     <ExpList><NameExp><IDName><ID><String>
         Input
     </String></ID></IDName></NameExp></ExpList>
   </CallExp>
</AssignmentStatement>
```

Figure 3.3: Example XML output of Guardol program

XML representation allows analysis, translation, and verification tools to work on Guardol while avoiding the need for these tools to work with internal representations in the Sireum framework. Currently the XML Builder is used to provide a path to Rockwell Collins ACL2 verification infrastructure. Up to this point in the project, this path has been used primarily to achieve a translation of a Guardol guard description to a Turnstile rule set.

Figure 3.3 presents an XML output of the translated simple Guardol code below.

```
virusOK := VIRUS_CHECK[Input];
```

3.4 IR Translation

The IR Translation phase translates a Guardol parse tree to an intermediate representation in the Sireum's Pilar modeling language. The Sireum framework provides a rich collection of static analysis and verification tools that work on the Pilar modeling language.

The Guardol Pilar IR is a three-address code representation. Three-address code is a common intermediate representation style that consists of a 4 tuple of: result, first operand, operator, second operand [6]. In three address code form, each instruction implements exactly one operation. For example, a complex statement such as:

```
X := A * B + C;
```

```
virusOK := VIRUS_CHECK[Input];
newBody := DIRTY_WORD_SEARCH[Input.body] when virusOk;
newBody := DIRTY_WORD_SEARCH[Input.body];
[newAttachments, attachAudit] := attachmentListCheck[Input.attachments];
Output := MIME_Type'(body => newBody, attachments => newAttachments);
Audit := attachAudit default
   (Dirty_Word_Check_Failed when (not exists newBody)) default
      (Virus_Check_Failed when (not virusOk));
```

Figure 3.4: A fragment of Guardol source code

```
# $temp1 := Input @Loc(45,18,45,22];
# call MIME::VIRUS_CHECK($temp1, $temp2) @Loc;
# virusOK @Loc := $temp2 @Loc;
# $temp4 := Input.body @Loc;
# call MIME::DIRTY_WORD_SEARCH($temp4, $temp5) @Loc;
# $temp6 := virusOk @Loc;
# $temp3 := when($temp5,$temp6) @Loc;
# newBody @Loc := $temp3 @Loc;
# $temp7 := Input.attachments @Loc;
# call MIME::attachmentListCheck($temp7, $temp8, $temp9) @Loc;
# newAttachments @Loc := $temp8 @Loc;
# attachAudit @Loc := $temp9 @Loc;
```

Figure 3.5: Pilar intermediate representation

is translated as two statements:

```
T := A * B;
X := T + C;
```

The form is usually employed in compiler frameworks to ease code analysis, transformation, and optimization because such meta-programs can assume simpler form of the code that they work with. Figure 3.4 presents the Guardol Pilar IR of the example in Figure 3.5.

The Pilar modeling language also features sophisticated annotation mechanism to store meta-data at various levels such as statements, expressions, and other programming language constructs. This facility allows, for example, code location information to be stored as meta-data, thus allowing mapping of Guardol Pilar IR code to the original Guardol code. Storing mapping information in the IR simplifies management of such information, and it eases debugging.

3.5 KSU Analysis and Verification

The analysis and verification module provides a collection of standard compiler-oriented data flow and control flow analyses, constraint-based type checking and inference, verification oriented analyses including modeling checking, symbolic execution and access to a variety of decision procedure packages. The current translation only makes use of the constraint-based type analysis to perform type checking for Guardol programs. In the next phase of the project, the analysis components will be utilized to a much greater extent.

3.6 Lustre Back-end

One of the primary tasks in the KSU Guardol Statement of Work requires translating to the Rockwell Collins representation of Lustre[7]. This provides a connection from Guardol to the Rockwell Collins Gryphon tool chain that provides a variety of model checking capabilities. The translation rules to the for the Guardol to Lustre translation are still being solidified. The translation framework is currently in place and being tested on the MIME example.

Chapter 4

Types

Guardol is a strongly typed language, requiring that all programs be well typed in accordance with the type constraint rules in Section 5.5. It is also statically typed, ensuring that all type checking is be completed before any translation or compilation to a target platform can be performed. This chapter will discuss the motivation behind the design decisions that lead to Guardol's type system followed by an in depth description the types themselves.

4.1 Motivation

In the construction of a DSL such as Guardol, an intuitive framework for the manipulation of structured data is paramount. A common approach to dealing with structured data within a language is the introduction of a type system. While the common use description of a type system is an expansive generalization on the specific goals we want to accomplish within the CDS domain, it does provide four functional areas of interest:

- Abstraction
- Safety
- Error Detection
- Documentation

which we will discuss in greater detail as they relate to the design of Guardol's own type system.

4.1.1 Abstraction

The nature of the CDS problem makes the use of data abstraction absolutely critical in the design of the Guardol language. Since the restricted domain of Guardol refers to the manipulation of "structured packets", there must be a mechanism within the language itself to specify the structure of these packets. Specifying the structure of data, in this case packets, is done by creating user defined types, as described in Section 4.3, or utilizing Guardol's primitive types, as described in Section 4.2. This is the first step towards creating a language framework that allows for the translation of security requirements into the implementation model of a CDS. This data abstraction layer enables a structural *interface* to exist between the domains of the system, where a domain's ability to communicate is limited by the type of the packets it can send to and receive from the CDS.

Guardol's type system enforces the structure of a packet by checking it against the specified type. A type cannot, however, be used to reason about the actual values contained within a given message. Given this, it is clear that not all security requirements can be assured by a lightweight type system; the introduction of property language and type qualifiers in future work would allow for a more expansive specification of security requirements that could be verified given an implementation model.

4.1.2 Safety

Naturally, since we are working to provide a DSL for the development of CDS solutions, the assurance of safety properties is a primary concern. The term "language safety" has a broad scope and can often be a point of contention when assertions are made that a certain language is "safe". For now, it will suffice to say that "a safe language is one that protects its own abstractions"[8]. As described in the above Section 4.1.1, the key abstractions utilized by Guardol's type system are those used to explicitly specify the structure of packets. The limitations placed on the specification of types and the construction of variables provide the user's of Guardol a high level of assurance that the implementation of a Guardol specification on any platform will only construct and transform packets in accordance with the specification.

4.1.3 Error Detection

As stated previously, Guardol is a language developed for the specification and implementation of CDS security policies. While many DSLs are designed to enable a faster development time, the nature of the security domain requires that security concerns take precedence over the simple elimination of burden placed on the user. This rationale, along with the realization that rapid prototyping is not a motivating factor in the design of Guardol, demanded the creation of a statically typed language. With a statically typed language, we are assured that no type errors will be raised during the execution of the guard on the target platform and that all type errors will be detected and corrected before the compilation of the guard.

4.1.4 Documentation

Natural language documentation often leaves something to be desired when used to describe the functionality of a program. It is difficult to coerce the ambiguity out and harder still to provide a clear and concise description of intent. Because Guardol was developed for both specification and implementation purposes, many attempts were made to merge the two whenever appropriate. Guardol type definitions provided an excellent opportunity to produce an unambiguous and readable form of specification. These types cannot only be used at an implementation level but they can also be used within the high level policy specification. In this way, the description of packet structure becomes rule governed and requires only that a superficial knowledge of the language be known to the reader.

4.2 Primitive Types

Guardol provides a simple collection primitive types - **bool**, **int**, **real**, **char**, and **string** -

- $[\![bool]\!] = \{\textbf{true}, \textbf{false}\}$
- $[\![int]\!] = \{\ldots, \textbf{-1}, \textbf{0}, \textbf{1}, \ldots\}$
- $[\![real]\!] = \mathbb{R}$
- $[\![char]\!]$ = UTF-8 charset
- $[\![string]\!] = [\![char]\!]^*$

that form the basis of communication between the CDS domains. Since these types are commonly used in almost all the conceivable target domains, we chose to formalize their use within Guardol's type system. Looking ahead, we expect to add a mechanism that allows primitive types to be customized to the target platform for which a guard is to be deployed. This will allow more of the guard's functionality to be specified within the Guardol program. For instance, some platforms may require that the specification of a **int** primitive refers to a 32-bit integer. These refinements would cause the generation of additional type constraints as described in Chapter 5.

Along with primitive types, Guardol also supports a number of operators (e.g. +, -, not) for use in the specification of an implementation model. This collection of arithmetic and boolean operators will provide Guardol users with the ability to model the computation implemented by the guard.

4.3 User Defined Types

User defined types provide Guardol with the ability to formalize the specific structure of incoming and outgoing packets within a guarding application. User defined types include a unique type name followed by a definition of the type's structure. This naming schema follows from similar approaches, e.g., in SPARK Ada, where the uniqueness of type names is a restriction motivated by general concerns in safety critical systems that include the need for increased clarity (sometimes at the expense of programmer effort) and the ability to trace each type to a particular definition. All type names are case sensitive.

```
package my_package is

type my_int is int;
type my_bool is bool;
type my_tuple is int * bool;

node external_node
  (i1 : my_int) returns
  (o1 : int, o2 : bool)
is external;

node internal_node
  (i1 : my_int, i2 : my_bool) returns
  (o1 : bool)
is
local
  t1 : my_tuple;
begin
  t1 := external_node(i1+2) when i2;
  o1 := t1#2;
end node;

end package;
```

Figure 4.1: Sample GIL code for demonstrating type constraints

User defined types are qualified by the package in which they are defined. They can be referenced within the defining package by type name (e.g. x : my_int) and referenced outside the defining package using the fully qualified type name (e.g. x : my_package.my_int). Because the enclosing package is used to qualify a type name outside of the package, type names are only required to be unique with respect to enclosing package's name space.

4.3.1 Simple Types

Simple type definitions take the general form

type *<type name>* **is** *<component type name>* ;

where a <*type name*> identifier represents a unique type name and <*component type name*> references an existing type. This type definition specifies a new type that is structurally equivalent to the component type used in its definition. These two types, however, are not considered equivalent in Guardol. The new simple type is a type alias of the component type, where as the new type may utilize all the functionality available to the component type but the two types are not compatible. The reasoning behind this distinction and a formal definition of type equivalence can be found in Section 4.5.

The following type definition in Figure 4.1 provides an example of a simple type definition,

```
type my_int is int;
```

where my_int is a newly defined type structurally identical to **int**. The shared structure of these two types is formally defined as $[\![int]\!]$.

4.3.2 List Types

List type definitions take the general form

type <*type name*> **is list** <*component type name*> **end list;**

where a <*type name*> identifier represents a unique type name and <*component type name*> references an existing type. This type definition specifies a new list type whose corresponding data structure is an ordered collection of values whose types are equivalent to the component type. A list is either empty, signified by **nil**, or a single value of the component type, the head, paired with another list which shares the type of the aforementioned list. This second list is known as the tail.

The manipulation of list types within the Guardol language is done using primitive recursion, that is, recursion that is numerically bounded such that it can only recurse a finite number of times. Though not currently implemented, the Guardol language will provide verification that requires all recursive calls to be performed on a substructure of the input parameters.

The following type definition in Figure 2.1 provides an example of a list type definition,

```
type stringList is list string end list;
```

where stringList is a newly defined list type whose component type is **string**.

4.3.3 Tuple Types

Tuple type definitions take the general form

> **type** <*type name*> **is**
>
> <*component type name 1*> *****
>
> . . . *****
>
> <*component type name n*> **;**

where a <*type name*> identifier represents a unique type name and each <*component type name*> references an existing type. This type definition specifies a new tuple type whose corresponding data structure consists of a value for each of the defined component types. Tuple data structures can be thought of as a simple ordered collection of values where each component type is paired with a single value in the collection. The values within a tuple data structure are accessed positionally based on the order in which their corresponding component type is specified in the tuple type definition.

Tuple types are the simplest way to define a structured data. Within Guardol tuples are the only form of structured data that do not require type names, though one can be assigned. Multi-variable assignments and node calls which return more than one variable implicitly create a tuple structure with no name composed positionally of the types corresponding the multiple variables. These unnamed tuples are compatible with any tuple with an identical structure. The following assignment provides an example of this compatibility

```
    t1 := external_node(i1+2) when i2;
```

where t1 is of type my tuple and external_node returns an unnamed tuple of whose structure is [**int** $*$ **bool**]. Tuple type equivalence is discussed in greater detail in Section 4.5.

The following type definition in Figure 4.1 provides an example of a tuple type definition,

```
type my_tuple is int * bool;
```

where my_tuple is a newly defined type whose component types are **int** and **bool**. A variable, v, of type my_tuple accesses its first component of type **int** using a positional reference to the first component, i.e. the expression v#1, and accesses its second component of type bool using a positional reference to the second component, i.e. the expression v#2. A type error is raised when a tuple access expression attempts to refer tuple component that is not defined.

4.3.4 Record Types

Record type definitions take the general form

> **type** <*type name*> **is record**
>
> <*field name 1*> : <*component type name 1*>**,**
>
> . . . **,**
>
> <*field name n*> : <*component type name n*>
>
> **end record;**

where a <*type name*> identifier represents a unique type name, each <*field name*> identifier represents a field name that is unique with respect to all other field names defined by the record type, and each <*component type name*> references an existing type. This type definition specifies a new record type whose corresponding data structure consists of a value for each of the defined component types. That is to say, a record data structure is a finite collection of field name and component value pairs where a single value exists for each component type and is accessed by the corresponding field name.

In Guardol, the advantages of using a record type over a tuple type lie in record type's ability to abstract over the position of its component types. The use of a field name to access a component of the record type enhances the clarity of the specification. However, since the record type abstracts over the position of component types, the Guardol conventions used in the positional construction and deconstruction tuples cannot be applied to records. In summation, we find that record types provide better specifications than tuple types but they have a more verbose definition, construction, and component access expression. For these reasons, we have included both type definitions in the Guardol language and defer to those designing Guardol programs to choose the best type for their purposes.

The following type definition in Figure 2.1 provides an example of a record type definition,

```
type Attachment is record
  name : string,
  object : AttachmentType
end record;
```

where Attachment is a newly defined record type whose component types are **string** and AttachmentType. A variable, v, of type Attachment accesses its first component of type **string** by referencing the field name name, i.e. the expression v.name, and accesses its second component of type AttachmentType by referencing the field name object, i.e. the expression v.object. A type error is raised when a record access expression attempts to refer record component that is not defined.

4.3.5 Union Types

Union type definitions take the general form

type <*type name*> **is union**

<*variant 1*> |

. . . |

<*variant n*>

end union;

where a <*type name*> identifier represents a unique type name and each <*variant*> takes the general form

<*union component name*> <*component type*>

or simply

<*union component name*>

Every <*variant*> contains a <*union component name*> which represents a unique name with respect to all other union component names defined in the union type and an optional <*component type*> which references an existing type. This type definition specifies a new union type that is a disjoint union of variants. A variant is composed of either a unique name or a unique name and a component type. This means that each variant is a place holder or inhabited by a value of the component type specified. The union data structure itself is inhabited by exactly one of the possible variants specified in the union type definition.

The following type definition in Figure 2.1 provides an example of a record type definition,

```
type AttachmentType is union
    MIME_Email of MIME_Type |
    Text of string |
    XML_DOM of XML_DOM_Type |
    Binary of ByteList |
    Other
end union;
```

where AttachmentType is a newly defined union type whose union component names are MIME_Email, Text, XML_DOM, Binary, and Other. A variable, v, of type AttachmentType is inhabited by exactly one of these union components. This definition provides an example of the two possible types of variants exemplified by MIME_Email and Other. If the variable v is inhabited by the union component MIME_Email then it also contains a value of type MIME_Type. Conversely, if the variable v is inhabited by the union component Other we know that the union component name itself provides the required information needed to reason about v.

4.4 Recursive and Mutually Recursive Types

Recursive type definitions, i.e., the use of the defining type name within the definition of that type name are restricted for use with union type definitions. Guardol also enables the definition of mutually recursive types, however this relationship must be explicitly stated through the use of the *andTypeDeclaration* (A.14). A mutually recursive type definition refers to two or more types that reference each other in their own type definitions. When defining mutually recursive types the first type definition is done in the standard fashion, while the subsequent type definitions replace the **type** keyword with the **and** keyword. All types connected together in this fashion have the capability to be mutually recursive. Types defined in this fashion must be sequentially defined within the Guardol language.

4.5 Type Equivalence

4.5.1 Overview

Type equivalence refers to the formal definition of compatibility between any two types within a type system. The two relevant approaches to type compatibility discussed during the design of Guardol were nominal and structural type compatibility. Each of these type compatibility models provides a unique approach to type equivalence and displays a number of different properties regarding the specification and use of data structures.

Nominative type compatibility, often used in procedural and object oriented languages, utilizes the definition of unique type names, where the name of a type is a first class representative of the type itself. Nominal type equivalence is performed by comparing two type names; if the type names are the same then the types are equivalent, otherwise they are not[6]. With the definition of new types, nominative type systems provide a simple way to produce type aliases that are structurally equivalent but not type equivalent. This is an important distinction that will be discussed in Subsection 4.5.6.

Structural type compatibility, as embodied in functional languages such as SML and OCAML, does not require that types be named. In a structural type system, “types are compatible if they have the same structure. To verify structural equivalence, user-defined type names are replaced by their definition. This process is repeated until no user-defined type name remains. The types are then considered structurally equivalent if they have exactly the same definition"[6]. Although the use of structural type equivalence is not limited to functional languages, structural type equivalence utilized in conjunction with a functional language provides a mechanism for type inference, i.e., a property of the type system whereby the structure of most types can be inferred. Using this model, type systems are able to achieve type safety properties without ever requiring the explicit specification of all type structures.

4.5.2 Motivation

To demonstrate the difference between a structural type system and the Guardol type system we will consider a number of potential assignments. First, we will define two types for use in these examples,

```
type my_bool is bool;
type my_tuple is int * bool;
```

For this following assignment, o1 is of the type **bool** and t1 is of type my_tuple

```
o1 := t1#2;
```

It can be shown that the expression t1#2 has type **bool** because it is a tuple access of t1's second component whose type is **bool**. This assignment statement is valid because the variable o1 and the assigned expression t1#2 are both of type **bool**.

Now, consider another potential assignment, where o1 is of type **bool** and i2 is of type my_bool

```
o1 := i2;
```

In this case, the assignment statement should not be valid because the variable o1 is of type **bool** and the assigned expression i2 is of type my_bool. Although these two types are structurally equivalent, they are not nominally compatible and therefore incompatible within the Guardol type system.

To further illustrate why this distinction is important, we will compare two similar functions in Guardol and SML. The following are two type definitions in Guardol.

```
type meters is int; type feet is int;
```

It is obvious from these two type names that we expect them to be incompatible with each other, even though we have that meters and feet are both structurally equivalent to $[\![int]\!]$. We do not want a variable x of type meters and a variable y of type feet to be used in the assignment statement

```
z := x + y;
```

no matter what the type of z.

In a language with a pure structural type system, such as SML, we must create a datatype to mimic the Guardol type definitions measurement

```
datatype measurement =
    METERS of int
  | FEET of int;
```

which is used to distinguish between the two types of measurement. However, in this case, it is assumed that variables of type measurement are opened upon execution and during the execution the guard must determine during runtime how to handle this discrepancy.

```
fun add (x,y) =
  case (x,y) of
    (METERS x1, METERS y1)
      => METERS (x1+y1)
  | (FEET x1, FEET y1)
      => FEET (x1+y1)
  | _ => raise IncompatibleMeasurementException;
```

4.5.3 Guardol Type Equivalence

The Guardol type system utilizes a hybrid approach of both nominal and structural type compatibility. Essentially, we want all types to be named, and we want type compatibility to be based on name compatibility. However, there are a few exceptional cases of structural type compatibility that we would like to allow as a convenience to the developer. These cases arise during the use of arithmetic literals and or when handling multiple return parameters from a node call.

For instance, given the assignment statement,

```
z := y + 7;
```

that z and y are both of the same type (e.g. feet), and the definition of that type is structurally equivalent to **int**, we would expect the type system to correctly type this assignment statement without the developer having to assign a type to the numeric literal 7. This requires that we do not incorporate a strictly nominal type system and that the structure of types are stored and evaluated against the primitive types of literals.

Second, when a node call has multiple return values, the grammar allows for the assignment statement to either split assignment of each tuple component individually or generate a tuple type which has defined component types but is not named. For example if the type my_tuple and the node external_node were defined as follows,

```
type my_tuple is int * bool;
node external_node
  (i1 : my_int) returns
  (o1 : int, o2 : bool)
is external;
```

we would want both of the following assignment statements to type check.

```
t1 := external_node(1);
o1,o2 :=external_node(2);
```

where t1 is of type my_tuple, o1 is of the type **int**, o2 is of the type **bool**, and external_node has the type structure [**int** $*$ **bool**].

To model the fact we want to include a degree of structural typing in the two situations above, we introduce the notion of type structure which exposes structure of primitive and tuple type instances generated by the type system such that they can be compared to user defined types. It is important to note that when a variable or parameter is declared with a primitive type, such as o1 in the code excerpt above, this is equivalent to mapping o1 to the type name **int** and not a type structure [**int**]. The constraint-based algorithm introduced in the following chapter generates these structures and uses them as internal representations of type structure. Formally, when referring to the structure of a types we must extend the type domain used in the type checking algorithm to include both type structures and type variables as follows

$$
\begin{array}{lcl}
\tilde{n} \in \mathit{qualifiedTypeName} & (A.11) & \\
\tilde{p} \in \mathit{primitiveType} & (A.16) & \\
\tilde{s} \in \mathit{typeStructure} & ::= & [\tilde{p}] \\
 & | & [\tilde{\tau}_1 * \ldots * \tilde{\tau}_n] \\
\tilde{\tau} \in \mathit{typeVariable} & ::= & \tilde{s} \\
 & | & \tilde{n}
\end{array}
$$

where a single primitive type name, e.g. [**bool**] or a tuple type, e.g. [**bool** $*$ my_int] is wrapped in single brackets to denote a type structure and a type variable denotes the union of type structures and qualified type names. The full semantics for Guardol type equivalence is defined in Figure 4.2. This semantics makes reference to the Guardol type environment defined in the following subsection.

4.5.4 Type Environment

The Guardol type environment, Σ, is a data structure used to maintain a mapping of user defined type names to their corresponding type definitions in the grammar excluding the **is** keyword. For example,

Σ[my_int $\mapsto$ **int**;
my_tuple $\mapsto$ **int** $*$ **bool**;
int_option $\mapsto$ **union** NONE | SOME **of int end union**]

would be a valid construction of the type environment with three user defined types. All user defined types are parsed and mapped into the type environment before any type equivalence is performed to ensure that the formalization for type equivalence holds. For use in the definition of type equivalence there is one lookup operation, $\Sigma(\tilde{\tau})$, that takes a type variable

$$\frac{}{\Sigma \vdash [\textsf{\textbf{bool}}] \simeq [\textsf{\textbf{bool}}]} \qquad \frac{}{\Sigma \vdash [\textsf{\textbf{int}}] \simeq [\textsf{\textbf{int}}]} \qquad \frac{}{\Sigma \vdash [\textsf{\textbf{real}}] \simeq [\textsf{\textbf{real}}]}$$

$$\frac{}{\Sigma \vdash [\textsf{\textbf{char}}] \simeq [\textsf{\textbf{char}}]} \qquad \frac{}{\Sigma \vdash [\textsf{\textbf{string}}] \simeq [\textsf{\textbf{string}}]} \qquad \frac{\Sigma \vdash \tilde{n}_1 = \tilde{n}_2}{\Sigma \vdash \tilde{n}_1 \simeq \tilde{n}_2}$$

$$\frac{\Sigma \vdash \Sigma(\tilde{n}) \simeq [\tilde{p}] \; when \; \Sigma(\tilde{n}) \neq \textsf{Fail}}{\Sigma \vdash \tilde{n} \simeq [\tilde{p}]} \qquad \frac{\Sigma \vdash \Sigma(\tilde{n}) \simeq [\tilde{\tau}_1 * \ldots * \tilde{\tau}_k] \; when \; \Sigma(\tilde{n}) \neq \textsf{Fail}}{\Sigma \vdash \tilde{n} \simeq [\tilde{\tau}_1 * \ldots * \tilde{\tau}_k]}$$

$$\frac{\Sigma \vdash \tilde{\tau}_1 \simeq \tilde{\tau}_2}{\Sigma \vdash \tilde{\tau}_2 \simeq \tilde{\tau}_1} \qquad \frac{\Sigma \vdash \tilde{\tau}_i^1 \simeq \tilde{\tau}_i^2 \quad i \in 1 \ldots k}{\Sigma \vdash [\tilde{\tau}_1^1 * \ldots * \tilde{\tau}_k^1] \simeq [\tilde{\tau}_1^2 * \ldots * \tilde{\tau}_k^2]}$$

Figure 4.2: Formal Definition of Type Equivalence

and returns a type structure or an indication of failure. If the lookup operation is performed on a type structure it will return that type structure, while a lookup operation performed on a type name that does not yield a type structure will return indication of failure. Figure 4.3 provides a formal definition of this lookup operation.

$$\frac{}{\Sigma(\textbf{bool}) = [\textbf{bool}]} \qquad \frac{}{\Sigma(\textbf{int}) = [\textbf{int}]} \qquad \frac{}{\Sigma(\textbf{real}) = [\textbf{real}]}$$

$$\frac{}{\Sigma(\textbf{char}) = [\textbf{char}]} \qquad \frac{}{\Sigma(\textbf{string}) = [\textbf{string}]} \qquad \frac{}{\Sigma(\tilde{s}) = \tilde{s}}$$

$$\frac{\Sigma[\tilde{n}_1 \mapsto \tilde{n}_2] \quad \Sigma(\tilde{n}_2) = \tilde{s}}{\Sigma(\tilde{n}_1) = \tilde{s}} \qquad \frac{\Sigma[\tilde{n}_1 \mapsto \tilde{n}_2] \quad \Sigma(\tilde{n}_2) = \mathsf{Fail}}{\Sigma(\tilde{n}_1) = \mathsf{Fail}} \qquad \frac{\Sigma[\tilde{n} \mapsto \tilde{n}_1 * \ldots * \tilde{n}_k]}{\Sigma(\tilde{n}) = [\tilde{n}_1 * \ldots * \tilde{n}_k]}$$

$$\frac{\Sigma[\tilde{n} \mapsto \textbf{list} \ldots]}{\Sigma(\tilde{n}) = \mathsf{Fail}} \qquad \frac{\Sigma[\tilde{n} \mapsto \textbf{record} \ldots]}{\Sigma(\tilde{n}) = \mathsf{Fail}} \qquad \frac{\Sigma[\tilde{n} \mapsto \textbf{union} \ldots]}{\Sigma(\tilde{n}) = \mathsf{Fail}}$$

Figure 4.3: Formal Definition of the Type Structure Lookup Operation

The type environment is also used in the evaluation of the type constraints from Chapter 5, where the data structure is identical to the one used in this chapter. Additional lookup operations on the type environment to expose the component types of type definitions are defined and explained in Section 5.2. These lookup operations are used to verify that a type name is mapped to a specific kind of composite type and/or return a specific component type name located in that composite type's definition. For example, the component type of a list type definition must be used to verify that the elements used to construct the list are of the correct type.

4.5.5 Example Derivations

Using the formal definitions from Figure 4.2 and Figure 4.3 we can formally derive the equivalence of two type variables used in some of the interesting examples in the above subsections.

Our first derivation will deal with the assignment statement

```
z := y + 7;
```

in which the type feet is equated to [**int**]. This derivation assumes that the type declaration

```
type feet is int;
```

is mapped into the type environment as follows $\Sigma[\mathsf{feet} \mapsto \textbf{int}]$. On the left, the type equivalence rules are used in conjunction with the type environment lookup rules, on the right, to derive the type equivalence of a type variable whose type is feet and the type structure [**int**].

$$\frac{\overline{\Sigma \vdash [\textbf{int}] \simeq [\textbf{int}]} \quad \textit{where } \Sigma(\mathsf{feet}) = [\textbf{int}]}{\Sigma \vdash \mathsf{feet} \simeq [\textbf{int}]} \qquad \frac{\Sigma[\mathsf{feet} \mapsto \textbf{int}] \quad \overline{\Sigma(\textbf{int}) = [\textbf{int}]}}{\Sigma(\mathsf{feet}) = [\textbf{int}]}$$

Our second derivation will deal with the assignment statement

```
t1 := external_node(1);
```

in which my_tuple is equated to [**int**$*$**bool**]. This derivation assumes that the type declaration

```
type my_tuple is int * bool;
```

is mapped into the type environment as follows $\Sigma[\mathsf{my\_tuple} \mapsto \textbf{int} * \textbf{bool}]$. Again, type equivalence rules and the type environment lookup rules are used to derive type equivalence of a type variable whose type is my_tuple and the type structure [**int** $*$ **bool**].

$$\frac{\dfrac{\overline{\Sigma \vdash [\textbf{int}] \simeq [\textbf{int}]} \quad \overline{\Sigma \vdash [\textbf{bool}] \simeq [\textbf{bool}]}}{\Sigma \vdash [\textbf{int} * \textbf{bool}] \simeq [\textbf{int} * \textbf{bool}] \ \textit{where } \Sigma(\mathsf{my\_tuple}) = [\textbf{int} * \textbf{bool}]}}{\Sigma \vdash \mathsf{my\_tuple} \simeq [\textbf{int} * \textbf{bool}]}$$

$$\frac{\Sigma[\mathsf{my\_tuple} \mapsto \textbf{int} * \textbf{bool}]}{\Sigma(\mathsf{my\_tuple}) = [\textbf{int} * \textbf{bool}]}$$

If my_tuple had the type structure [**int** $*$ **bool**], no derviation would be possible. This would indicate that a type equivalence could not be constructed.

4.5.6 Assessment

A purely structural type compatibility was disregarded based on its inability to complement the language design goals specified for Guardol, and a purely nominative type compatibility was abandoned due to the disparities that arise with the introduction of tuples that are not uniquely named. This assessment lead to the development of a new set of type equivalence for use in the Guardol language. The key design features that guided its development were

- A correspondence between semantic domains where the exchange of values between domains requires explicit conversions, e.g., feet and meters and
- A type specification such that the details of a type implementation are hidden from the clients of that type.

A nominal element of type equivalence is needed to satisfy both of these requirements and a structural element of type equivalence is needed to incorporate tuple compatibility. Guardol's hybrid type system, while possibly placing a small additional burden on user with respect to a pure structural type system, provides

- name-centric type specification – type names that correspond to semantic domains
- enhanced domain specification readability and verification
- static checks for incompatible types that are structurally identical before the execution of the guard
- elimination of case statements describing the compatibility of user defined primitive types under their respective operators

coinciding with Guardol's language design goals.

Chapter 5

Type Constraints

After the initial parsing and symbol resolution phases, the Guardol compiler carries out a type checking phase to enforce compliance to the Guardol type system. Type checking is implemented via a constraint solving algorithm. Constraints are generated first in a syntax-directed traversal of the Guardol program's AST, then the constraints are solved using an unification algorithm.

Specifically, constraint generation begins by compiling the primitive and user defined types into a type environment, Σ, and the user defined nodes into a node environment, Ψ. As the list of type constraints is generated and evaluated based on the type constraint generation rules in Section 5.5, a cache of type variables, expression labels, and variable names is denoted by $\hat{C}$. In Guardol, there is no notion of a "global variable" – all variable references refer to a local variable or parameter, so there is a unique $\hat{C}$ for each node such any type constraint generated for a node does not impact other nodes.

After the type checking phase is finished and there are no errors raised, the type checking phase is considered complete. An error in the type checking indicates the usage of an undefined type, a type mismatch, an incorrect type construction, or an incorrect type access.

5.1 Syntax Domain

The type constraint generation rules below follow the structure of the grammar in Section A. However, for conciseness in stating the type constraint rules, we abbreviate the labels of nonterminals and terminals used in Appendix A. Figure 5.1 summarizes these abbreviations. Exp-Labels, or expression labels, comprise the only additional syntax domain extension found within these abbreviations. An expression label is simply an unique label used to identify each individual expression AST node. The extension is required to formalize the type constraint generation rules in Section 5.5.

$$
\begin{array}{lll}
\bar{b} \in [\![\mathit{bool}]\!] & \bar{d} \in \mathit{constructorName}\ (\mathrm{A.5}) & \bar{e} \in \mathit{expression}\ (\mathrm{A.30}) \\
\bar{c} \in [\![\mathit{char}]\!] & \bar{f} \in \mathit{fieldName}\ (\mathrm{A.8}) & \bar{l} \in \text{Exp-Labels} \\
\bar{i} \in [\![\mathit{int}]\!] & \bar{m} \in \mathit{statement}\ (\mathrm{A.62}) & \overline{ms} \in \mathit{statementList}\ (\mathrm{A.61}) \\
\bar{r} \in [\![\mathit{real}]\!] & \bar{o} \in \mathit{qualifiedNodeName}\ (\mathrm{A.10}) & \bar{p} \in \mathit{pattern}\ (\mathrm{A.71}) \\
\bar{s} \in [\![\mathit{string}]\!] & \bar{x} \in \mathit{variableName}\ (\mathrm{A.6}) & \overline{var} \in \bar{x} \cup \bar{l} \cup \tilde{\tau}
\end{array}
$$

$$
\begin{array}{ll}
\overline{n\_op} \in & \mathit{binaryNumericOp1}\ (\mathrm{A.49}) \cup \mathit{binaryNumericOp2}\ (\mathrm{A.49}) \\
& \qquad \cup\ \mathit{binaryRelationalOp}\ (\mathrm{A.53}) \\
\overline{b\_op} \in \mathit{binaryBooleanOp}\ (\mathrm{A.55}) & \overline{u\_op} \in \mathit{unaryNumericOp}\ (\mathrm{A.45})
\end{array}
$$

Figure 5.1: Type Constraint Syntax Domain

5.2 Data Structures

The type checking algorithm utilizes three data structures Σ, Ψ, and $\hat{C}$ in the process of constraint generation and solving. The table below summarizes these data structures and indicates the output of the lookup operations used to access their contents. Pass and Fail are terms introduced at this stage to identify whether or not a lookup operation could be completed. Fail is returned when a lookup operation attempts to access a node or particular kind of type definition that is not defined in the environment according to the lookup operations parameters. If the lookup operation does not return Fail then it will return a type variable, a type structure, or an affirmative Pass which signifies that this type or node definedness portion of the constraint rule is satisfied.

(types)	Σ	Type Environment
	$\Sigma(\tilde{\tau})$	Type Variable $\rightarrow$ Type Structure $\cup$ Fail
	$\Sigma_{listComponent}(\tilde{\tau})$	Type Variable $\rightarrow$ Type Variable $\cup$ Fail
	$\Sigma_{recordComponent}(\tilde{\tau}, \bar{f})$	Type Variable * Field Name $\rightarrow$ Type Variable $\cup$ Fail
	$\Sigma_{tupleComponent}(\tilde{\tau}, i)$	Type Variable * $\mathbb{N}$ $\rightarrow$ Type Variable $\cup$ Fail
	$\Sigma_{unionComponent}(\tilde{\tau}, \bar{d})$	Type Variable * Constructor Name $\rightarrow$ Type Variable $\cup$ Fail
	$\Sigma_{isList}(\tilde{\tau})$	Type Variable $\rightarrow$ Pass $\cup$ Fail
	$\Sigma_{hasComponent}(\tilde{\tau}, \bar{d})$	Type Variable * Constructor Name $\rightarrow$ Pass $\cup$ Fail
	$\Sigma_{hasConstructor}(\tilde{\tau}, \bar{d})$	Type Variable * Constructor Name $\rightarrow$ Pass $\cup$ Fail
(nodes)	Ψ	Node Environment
	$\Psi_{in}(\bar{o}, i)$	Qualified Node Name * $\mathbb{N}$ $\rightarrow$ Type Variable $\cup$ Fail
	$\Psi_{out}(\bar{o}, i)$	Qualified Node Name * $\mathbb{N}$ $\rightarrow$ Type Variable $\cup$ Fail
(cache)	$\hat{C}$	Type Variable Cache
	$\hat{C}(\overline{var})$	Variable Name $\cup$ Exp-Labels $\cup$ Type Variable
		$\rightarrow$ Variable Name $\cup$ Exp-Labels $\cup$ Type Variable

A brief description of each of the above lookup operations is provided below.

- $\Sigma(\tilde{\tau})$ - Given any type variable, this lookup operation returns the type structure associated with it or an indication of failure if there is no type structure associated with this type variable. This operation is described in more depth in Section 4.5.4.

- $\Sigma_{listComponent}(\tilde{\tau})$ - Given a type variable that is of the list type, this lookup operation returns the component type variable. An indication of failure is returned if the given type variable is of any other type or is not defined.

- $\Sigma_{recordComponent}(\tilde{\tau}, \bar{f})$ - Given a type variable that is of the record type, this lookup operation returns the component type variable associated with the field $\bar{f}$. An indication of failure is returned if the given type variable is of any other type, is not defined, or the given field does not exist.

- $\Sigma_{tupleComponent}(\tilde{\tau}, i)$ - Given a type variable that is of the tuple type, this lookup operation returns the component type variable associated with the index i. An indication of failure is returned if the given type variable is of any other type, is not defined, or the given index does not exist.

- $\Sigma_{unionComponent}(\tilde{\tau}, \bar{d})$ - Given a type variable that is of the union type, this lookup operation returns the component type variable associated with the constructor $\bar{d}$. An indication of failure is returned if the given type variable is of any other type, is not defined, the given constructor does not exist, or the given constructor is not associated with a type.

- $\Sigma_{isList}(\tilde{\tau})$ - Given a type variable that is of the list type, this lookup operation returns an affirmative indication. An indication of failure is returned if the given type variable is of any other type or is not defined.

- $\Sigma_{hasComponent}(\tilde{\tau}, \bar{d})$ - Given a type variable that is of the union type, this lookup operation returns an affirmative indication if the constructor $\bar{d}$ has a component type associated with it. An indication of failure is returned if the given type variable is of any other type, is not defined, or the given constructor does not exist.

- $\Sigma_{hasConstructor}(\tilde{\tau}, \bar{d})$ - Given a type variable that is of the union type, this lookup operation returns an affirmative indication if the constructor $\bar{d}$ exists. An indication of failure is returned if the given type variable is of any other type, is not defined, or the given constructor does not exist.

- $\Psi_{in}(\bar{o}, i)$ - Given a node name $\bar{o}$, this lookup returns a type variable of its i^{th} input parameter. An indication of failure is returned if the node is not defined or does not have an i^{th} input parameter.

- $\Psi_{out}(\bar{o}, i)$ - Given a node name $\bar{o}$, this lookup returns a type variable of its i^{th} output parameter. An indication of failure is returned if the node is not defined or does not have an i^{th} output parameter.

- The cache $\hat{C}$ is the primary data structure used in the constraint solving algorithm. Each use of $\hat{C}(\overline{var})$ represents a call of the FIND operation defined in Section 5.6.

5.3 Definedness Extension of Type Equivalence

The definition of type equivalence in Figure 4.2 does not account for the possibility that the lookup of a type variable in the type environment can fail. Figure 5.2 introduces a new judgement, $\Sigma \vdash \tilde{\tau}_1 \simeq_{\downarrow} \tilde{\tau}_2$, that can only hold if all the type variables occurring in the given types are also present in Σ. According to these rules, all primitive type structures are automatically defined and tuple type structures are defined if and only if their corresponding component type variables are defined. Type names are defined if the type environment lookup does not return a Fail condition. The final rule constructs a new type equivalence relation by which the two type variables must be equivalent based on the previous type equivalence formalization and each type variable must be defined based on the definedness formalization.

$$\frac{}{\Sigma \vdash\downarrow [\textbf{bool}]} \qquad \frac{}{\Sigma \vdash\downarrow [\textbf{int}]} \qquad \frac{}{\Sigma \vdash\downarrow [\textbf{real}]} \qquad \frac{}{\Sigma \vdash\downarrow [\textbf{char}]}$$

$$\frac{}{\Sigma \vdash\downarrow [\textbf{string}]} \qquad \frac{\Sigma \vdash\downarrow \tilde{\tau}_i \quad i \in 1 \ldots k}{\Sigma \vdash\downarrow [\tilde{\tau}_1 * \ldots * \tilde{\tau}_k]} \qquad \frac{\text{Fail} \notin \Sigma(\tilde{n})}{\Sigma \vdash\downarrow \tilde{n}}$$

$$\frac{\Sigma \vdash \tilde{\tau}_1 \simeq \tilde{\tau}_2 \quad \downarrow \tilde{\tau}_1 \quad \downarrow \tilde{\tau}_2}{\Sigma \vdash \tilde{\tau}_1 \simeq_{\downarrow} \tilde{\tau}_2}$$

Figure 5.2: Definedness Extension of Type Equivalence

5.4 Rule Form

As an example of the general form for constraint rules consider the rule for multiple assignments.

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_S \bar{x}_1\texttt{,}\ \ldots\texttt{,}\ \bar{x}_n \texttt{ := } \bar{e}^{\bar{l}} \quad \text{iff} \quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}^{\bar{l}}$$
$$\text{and} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} [\hat{C}(\bar{x}_1) * \ldots * \hat{C}(\bar{x}_n)]$$

The $\langle \hat{C},\ \Sigma,\ \Psi\rangle$ represents the context for the constraint; it indicates the current auxiliary data structures used to solve each constraint. The rule should be understood as follows in the context of the data structures $\langle \hat{C},\ \Sigma,\ \Psi\rangle$, $\bar{x}_n$ **:=** $\bar{e}^{\bar{l}}$ is correctly typed if and only if $\bar{e}^{\bar{l}}$ is correctly typed in the same context and if the type $\hat{C}(\bar{l})$ of the expression $\bar{e}$ (which is found by looking up the expression label l of $\bar{e}$ in the cache) is a tuple type where each component type of the matches the type $\hat{C}(\bar{x}_i)$ of the corresponding variable $\bar{x}_i$.

Each constraint generation rule follows the structure of the associated grammar rule in Section A. Algorithmically, constraints are generated in a recursive traversal of the parse tree. The arguments of the function carrying out the traversal are the data structures from the context $\langle \hat{C},\ \Sigma,\ \Psi\rangle$ and the current AST node (e.g. $\bar{x}_1\texttt{,}\ \ldots\texttt{,}\ \bar{x}_n \texttt{ := } \bar{e}^{\bar{l}}$). The function will recursively call the traversal algorithm for subcomponents (e.g. $\bar{e}^{\bar{l}}$), and will add constraints representing type equality/equivalence conditions that need to be enforced for type soundness.

As described in the above Section 5.3 type equivalence is extended to reason about the definedness of the namespace including but not limited to type names, field names, variable names, node names, and parameter names. Each type constraint generation rule follows one of two different forms. A constraint is either an equality constraint, $\simeq_{\downarrow}$, where two type variables are equated together, in this case $\hat{C}(\bar{l}) \simeq_{\downarrow} [\hat{C}(\bar{x}_1) * \ldots * \hat{C}(\bar{x}_n)]$, or a strict definedness constraint that returns a Pass or Fail. A constraint rule is not satisfied it generates a Fail condition. Equality constraints that cannot be satisfied indicate a type checking error.

5.5 Type Constraint Rules

The following type constraint rules are generated as the Abstract Syntax Tree (AST) is traversed. Since not all nodes of the AST generate type constraints, only those relevant to the type system are enumerated in the following type constraint rules. For example, although there is no specific constraint generation rule for *nodeDeclaration* (A.6), we will assume that all parameter variable name declarations and local variable name declarations will be visited sequentially followed by the statements defined within the node. We will also assume that both the type environment, Σ, and the node environment, Ψ, have been constructed with all the definitions required to verify the typing rules. The cache, $\hat{C}$, is assumed to be empty when a new node is visited.

$$\langle \hat{C},\ \Sigma,\ \Psi \rangle \models_D \bar{x} \textbf{ : } \tilde{n} \quad \text{iff} \quad \hat{C}(\bar{x}) \simeq_\downarrow \tilde{n} \tag{5.1}$$

$$\langle \hat{C},\ \Sigma,\ \Psi \rangle \models_D \bar{x} \textbf{ : } \tilde{n}_1 * \ldots * \tilde{n}_n \quad \text{iff} \quad \hat{C}(\bar{x}) \simeq_\downarrow [\tilde{n}_1 * \ldots * \tilde{n}_n] \tag{5.2}$$

$$\langle \hat{C},\ \Sigma,\ \Psi \rangle \models_S \bar{x} \textbf{ := } \bar{e}^{\bar{l}} \quad \text{iff} \quad \langle \hat{C},\ \Sigma,\ \Psi \rangle \models_E \bar{e}^{\bar{l}} \quad \text{and} \quad \hat{C}(\bar{l}) \simeq_\downarrow \hat{C}(\bar{x}) \tag{5.3}$$

$$\langle \hat{C},\ \Sigma,\ \Psi \rangle \models_S \bar{x}_1 \textbf{, } \ldots \textbf{, } \bar{x}_n \textbf{ := } \bar{e}^{\bar{l}} \quad \text{iff} \quad \langle \hat{C},\ \Sigma,\ \Psi \rangle \models_E \bar{e}^{\bar{l}} \quad \text{and} \quad \hat{C}(\bar{l}) \simeq_\downarrow [\hat{C}(\bar{x}_1) * \ldots * \hat{C}(\bar{x}_n)] \tag{5.4}$$

$$\langle \hat{C},\ \Sigma,\ \Psi \rangle \models_S \textbf{if } \bar{e}^{\bar{l}} \textbf{ then } \overline{ms} \quad \text{iff} \quad \langle \hat{C},\ \Sigma,\ \Psi \rangle \models_E \bar{e}^{\bar{l}} \quad \text{and} \quad \hat{C}(\bar{l}) \simeq_\downarrow [\textbf{bool}] \quad \text{and} \quad \langle \hat{C},\ \Sigma,\ \Psi \rangle \models_{SS} \overline{ms} \tag{5.5}$$

$$\langle \hat{C},\ \Sigma,\ \Psi \rangle \models_S \textbf{elseif } \bar{e}^{\bar{l}} \textbf{ then } \overline{ms} \quad \text{iff} \quad \langle \hat{C},\ \Sigma,\ \Psi \rangle \models_E \bar{e}^{\bar{l}} \quad \text{and} \quad \hat{C}(\bar{l}) \simeq_\downarrow [\textbf{bool}] \quad \text{and} \quad \langle \hat{C},\ \Sigma,\ \Psi \rangle \models_{SS} \overline{ms} \tag{5.6}$$

$$\langle \hat{C},\ \Sigma,\ \Psi \rangle \models_S \textbf{else } \overline{ms} \quad \text{and} \quad \langle \hat{C},\ \Sigma,\ \Psi \rangle \models_{SS} \overline{ms} \tag{5.7}$$

$$\langle \hat{C},\ \Sigma,\ \Psi \rangle \models_S \textbf{match } \bar{e}^{\bar{l}} \textbf{ with} \quad \text{iff} \quad \langle \hat{C},\ \Sigma,\ \Psi \rangle \models_E \bar{e}^{\bar{l}} \tag{5.8}$$

$$
\begin{array}{lll}
& \bar{p}_1 \mathbf{=>}\ \overline{ms}_1 \quad & \text{and}\quad \forall i.\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_{SS}\ \overline{ms}_i \\
& \big|\ldots\big|\bar{p}_n \mathbf{=>}\ \overline{ms}_n & \text{and}\quad \forall i.\langle \hat{C},\ \Sigma,\ \Psi,\ \hat{C}(\bar{l})\rangle \models_{P}\ \bar{p}_i \\
\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_{SS}\ \bar{m}\,\mathbf{;} & & \text{and}\quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_{S}\ \bar{m} \\
\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_{SS}\ \bar{m}\,\mathbf{;}\,\overline{ms} & & \text{and}\quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_{S}\ \bar{m} \\
& & \text{and}\quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_{SS}\ \overline{ms}
\end{array}
$$

$$\langle \hat{C},\ \Sigma,\ \Psi,\ \tilde{\tau}\rangle \models_P\ \bar{x} \quad \text{iff}\quad \hat{C}(\bar{x}) \simeq_{\downarrow} \tilde{\tau} \tag{5.9}$$

$$\langle \hat{C},\ \Sigma,\ \Psi,\ \tilde{\tau}\rangle \models_P\ \_ \tag{5.10}$$

$$\langle \hat{C},\ \Sigma,\ \Psi,\ \tilde{\tau}\rangle \models_P\ \textbf{nil} \quad \text{iff}\quad \Sigma_{isList}(\tilde{\tau}) \tag{5.11}$$

$$
\begin{array}{lll}
\langle \hat{C},\ \Sigma,\ \Psi,\ \tilde{\tau}\rangle \models_P\ \bar{x}_1\ \mathbf{::}\ \bar{x}_2 & \text{iff} & \Sigma_{isList}(\tilde{\tau}) \\
& \text{and} & \hat{C}(\bar{x}_1) \simeq_{\downarrow} \Sigma_{listComponent}(\tilde{\tau}) \\
& \text{and} & \hat{C}(\bar{x}_2) \simeq_{\downarrow} \tilde{\tau}
\end{array} \tag{5.12}
$$

$$
\begin{array}{lll}
\langle \hat{C},\ \Sigma,\ \Psi,\ \tilde{\tau}\rangle \models_P\ \_\ \mathbf{::}\ \bar{x}_2 & \text{iff} & \Sigma_{isList}(\tilde{\tau}) \\
& \text{and} & \hat{C}(\bar{x}_2) \simeq_{\downarrow} \tilde{\tau}
\end{array} \tag{5.13}
$$

$$
\begin{array}{lll}
\langle \hat{C},\ \Sigma,\ \Psi,\ \tilde{\tau}\rangle \models_P\ \bar{x}_1\ \mathbf{::}\ \_ & \text{iff} & \Sigma_{isList}(\tilde{\tau}) \\
& \text{and} & \hat{C}(\bar{x}_1) \simeq_{\downarrow} \Sigma_{listComponent}(\tilde{\tau})
\end{array} \tag{5.14}
$$

$$
\begin{array}{lll}
\langle \hat{C},\ \Sigma,\ \Psi,\ \tilde{\tau}\rangle \models_P\ \bar{d}\,\bar{x} & \text{iff} & \Sigma_{hasComponent}(\tilde{\tau}, \bar{d}) \\
& \text{and} & \Sigma_{hasConstructor}(\tilde{\tau}, \bar{d}) \\
& \text{and} & \hat{C}(\bar{x}) \simeq_{\downarrow} \Sigma_{unionComponent}(\tilde{\tau}, \bar{d})
\end{array} \tag{5.15}
$$

$$
\begin{array}{lll}
\langle \hat{C},\ \Sigma,\ \Psi,\ \tilde{\tau}\rangle \models_P\ \bar{d}\ \_ & \text{iff} & \Sigma_{hasComponent}(\tilde{\tau}, \bar{d}) \\
& \text{and} & \Sigma_{hasConstructor}(\tilde{\tau}, \bar{d})
\end{array} \tag{5.16}
$$

$$\langle \hat{C},\ \Sigma,\ \Psi,\ \tilde{\tau}\rangle \models_P\ \bar{d} \quad \text{iff}\quad \Sigma_{hasConstructor}(\tilde{\tau}, \bar{d}) \tag{5.17}$$

$$
\begin{array}{lll}
\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E\ (\bar{o}\boldsymbol{(}\bar{e}_1^{\bar{l}_1}\boldsymbol{,}\ldots\boldsymbol{,}\bar{e}_n^{\bar{l}_n}\boldsymbol{)})^{\bar{l}} & \text{iff} & \forall i.\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E\ \bar{e}_i^{\bar{l}_i} \\
& \text{and} & \forall i.\hat{C}(\bar{l}_i) \simeq_{\downarrow} \Psi_{in}(\bar{o}, i) \\
& \text{and} & \hat{C}(\bar{l}) \simeq_{\downarrow} \Psi_{out}(\bar{o}, 1)\ \big|\ n = 1
\end{array} \tag{5.18}
$$

$$\text{and}\quad \hat{C}(\bar{l}) \simeq_{\downarrow} [\Psi_{out}(\bar{o},1) * \ldots * \Psi_{out}(\bar{o},n)] \;\big|\; n > 1$$

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{x}^{\bar{l}} \quad \text{iff} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} \hat{C}(\bar{x}) \tag{5.19}$$

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{b}^{\bar{l}} \quad \text{iff} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} [\textsf{\textbf{bool}}] \tag{5.20}$$

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{i}^{\bar{l}} \quad \text{iff} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} [\textsf{\textbf{int}}] \tag{5.21}$$

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{r}^{\bar{l}} \quad \text{iff} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} [\textsf{\textbf{real}}] \tag{5.22}$$

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{c}^{\bar{l}} \quad \text{iff} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} [\textsf{\textbf{char}}] \tag{5.23}$$

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{s}^{\bar{l}} \quad \text{iff} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} [\textsf{\textbf{string}}] \tag{5.24}$$

$$\begin{array}{lll}
\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E (\bar{e}_1^{\bar{l}_1}\,\boldsymbol{\#}n)^{\bar{l}} & \text{iff} & \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_1^{\bar{l}_1} \\
 & \text{and} & \hat{C}(\bar{l}) \simeq_{\downarrow} \Sigma_{tupleComponent}(\hat{C}(\bar{l}_1), n)
\end{array} \tag{5.25}$$

$$\begin{array}{lll}
\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E (\bar{e}_1^{\bar{l}_1}\boldsymbol{.}\bar{f})^{\bar{l}} & \text{iff} & \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_1^{\bar{l}_1} \\
 & \text{and} & \Sigma_{recordComponent}(\hat{C}(\bar{l}_1), \bar{f}) \simeq_{\downarrow} \hat{C}(\bar{l})
\end{array} \tag{5.26}$$

$$\begin{array}{lll}
\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E (\tilde{n}\boldsymbol{'(}\bar{e}_1^{\bar{l}_1}\boldsymbol{,}\ldots\boldsymbol{,}\bar{e}_n^{\bar{l}_n}\boldsymbol{))}^{\bar{l}} & \text{iff} & \forall i.\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_i^{\bar{l}_i} \\
 & \text{and} & \hat{C}(\bar{l}) \simeq_{\downarrow} \tilde{n} \\
 & \text{and} & \forall i.\Sigma_{tupleComponent}(\tilde{n}, i) \simeq_{\downarrow} \hat{C}(\bar{l}_i)
\end{array} \tag{5.27}$$

$$\begin{array}{lll}
\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E (\tilde{n}\boldsymbol{'}\{\bar{e}_1^{\bar{l}_1}\boldsymbol{,}\ldots\boldsymbol{,}\bar{e}_n^{\bar{l}_n}\})^{\bar{l}} & \text{iff} & \forall i.\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_i^{\bar{l}_i} \\
 & \text{and} & \hat{C}(\bar{l}) \simeq_{\downarrow} \tilde{n} \\
 & \text{and} & \forall i.\Sigma_{listComponent}(\tilde{n}) \simeq_{\downarrow} \hat{C}(\bar{l}_i)
\end{array} \tag{5.28}$$

$$\begin{array}{lll}
\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E (\tilde{n}\boldsymbol{'[}\bar{f}_1\boldsymbol{=}>\bar{e}_1^{\bar{l}_1} & \text{iff} & \forall i.\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_i^{\bar{l}_i} \\
\qquad\qquad\qquad\qquad\boldsymbol{,}\ldots\boldsymbol{,} & \text{and} & \hat{C}(\bar{l}) \simeq_{\downarrow} \tilde{n} \\
\qquad\qquad\qquad\qquad\bar{f}_n\boldsymbol{=}>\bar{e}_n^{\bar{l}_n}\boldsymbol{]})^{\bar{l}} & \text{and} & \forall i.\Sigma_{recordComponent}(\tilde{n}, \bar{f}_i) \simeq_{\downarrow} \hat{C}(\bar{l}_i)
\end{array} \tag{5.29}$$

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E (\tilde{n}\boldsymbol{'}\bar{d})^{\bar{l}} \quad \text{iff} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} \tilde{n} \tag{5.30}$$

$$\begin{array}{lll}
\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E (\tilde{n}\boldsymbol{'}\bar{d}\boldsymbol{(}\bar{e}_1^{\bar{l}_1}\boldsymbol{)})^{\bar{l}} & \text{iff} & \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_1^{\bar{l}_1} \\
 & \text{and} & \hat{C}(\bar{l}) \simeq_{\downarrow} \tilde{n} \\
 & \text{and} & \Sigma_{unionComponent}(\tilde{n}, \bar{d}) \simeq_{\downarrow} \hat{C}(\bar{l}_1)
\end{array} \tag{5.31}$$

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E (\textbf{when(}\bar{e}_1^{\bar{l}_1}\textbf{, } \bar{e}_2^{\bar{l}_2}\textbf{)})^{\bar{l}} \quad \text{iff} \quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_1^{\bar{l}_1} \tag{5.32}$$

$$\text{and} \quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_2^{\bar{l}_2}$$

$$\text{and} \quad \hat{C}(\bar{l}_2) \simeq_{\downarrow} [\textbf{bool}]$$

$$\text{and} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} \hat{C}(\bar{l}_1)$$

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E (\textbf{default(}\bar{e}_1^{\bar{l}_1}\textbf{, } \bar{e}_2^{\bar{l}_2}\textbf{)})^{\bar{l}} \quad \text{iff} \quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_1^{\bar{l}_1} \tag{5.33}$$

$$\text{and} \quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_2^{\bar{l}_2}$$

$$\text{and} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} \hat{C}(\bar{l}_2)$$

$$\text{and} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} \hat{C}(\bar{l}_1)$$

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E (\textbf{exist(}\bar{e}_1^{\bar{l}_1}\textbf{)})^{\bar{l}} \quad \text{iff} \quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_1^{\bar{l}_1} \tag{5.34}$$

$$\text{and} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} [\textbf{bool}]$$

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E (\bar{e}_1^{\bar{l}_1}\textbf{::}\bar{e}_2^{\bar{l}_2})^{\bar{l}} \quad \text{iff} \quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_1^{\bar{l}_1} \tag{5.35}$$

$$\text{and} \quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_2^{\bar{l}_2}$$

$$\text{and} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} \hat{C}(\bar{l}_2)$$

$$\text{and} \quad \Sigma_{listComponent}(\hat{C}(\bar{l}_2)) \simeq_{\downarrow} \hat{C}(\bar{l}_1)$$

$$\text{and} \quad \Sigma_{isList}(\hat{C}(\bar{l}_2))$$

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E (\bar{e}_1^{\bar{l}_1}\textbf{@}\bar{e}_2^{\bar{l}_2})^{\bar{l}} \quad \text{iff} \quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_1^{\bar{l}_1} \tag{5.36}$$

$$\text{and} \quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_2^{\bar{l}_2}$$

$$\text{and} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} \hat{C}(\bar{l}_2)$$

$$\text{and} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} \hat{C}(\bar{l}_1)$$

$$\text{and} \quad \Sigma_{isList}(\hat{C}(\bar{l}_2))$$

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E (\textbf{not } \bar{e}_1^{\bar{l}_1})^{\bar{l}} \quad \text{iff} \quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_1^{\bar{l}_1} \tag{5.37}$$

$$\text{and} \quad \hat{C}(\bar{l}) \simeq_{\downarrow} \hat{C}(\bar{l}_1)$$

$$\text{and} \quad \hat{C}(\bar{l}_1) \simeq_{\downarrow} [\textbf{bool}]$$

$$\langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E (\overline{u\_op}\ \bar{e}_1^{\bar{l}_1})^{\bar{l}} \quad \text{iff} \quad \langle \hat{C},\ \Sigma,\ \Psi\rangle \models_E \bar{e}_1^{\bar{l}_1} \tag{5.38}$$

$$
\begin{array}{lll}
 & \text{and} & \hat{C}(\bar{l}) \simeq_{\downarrow} \hat{C}(\bar{l_1}) \\
 & \text{and} & \hat{C}(\bar{l_1}) \simeq_{\downarrow} [\textbf{int}] \\
\langle \hat{C},\ \Sigma,\ \Psi \rangle \ \models_E \ (\bar{e}_1^{\bar{l}_1}\ \overline{b\_op}\ \bar{e}_2^{\bar{l}_2})^{\bar{l}} & \text{iff} & \langle \hat{C},\ \Sigma,\ \Psi \rangle \ \models_E \ \bar{e}_1^{\bar{l}_1} \qquad (5.39) \\
 & \text{and} & \langle \hat{C},\ \Sigma,\ \Psi \rangle \ \models_E \ \bar{e}_2^{\bar{l}_2} \\
 & \text{and} & \hat{C}(\bar{l}) \simeq_{\downarrow} \hat{C}(\bar{l_1}) \\
 & \text{and} & \hat{C}(\bar{l_1}) \simeq_{\downarrow} \hat{C}(\bar{l_2}) \\
 & \text{and} & \hat{C}(\bar{l_2}) \simeq_{\downarrow} [\textbf{bool}] \\
\langle \hat{C},\ \Sigma,\ \Psi \rangle \ \models_E \ (\bar{e}_1^{\bar{l}_1}\ \overline{n\_op}\ \bar{e}_2^{\bar{l}_2})^{\bar{l}} & \text{iff} & \langle \hat{C},\ \Sigma,\ \Psi \rangle \ \models_E \ \bar{e}_1^{\bar{l}_1} \qquad (5.40) \\
 & \text{and} & \langle \hat{C},\ \Sigma,\ \Psi \rangle \ \models_E \ \bar{e}_2^{\bar{l}_2} \\
 & \text{and} & \hat{C}(\bar{l}) \simeq_{\downarrow} \hat{C}(\bar{l_1}) \\
 & \text{and} & \hat{C}(\bar{l_1}) \simeq_{\downarrow} \hat{C}(\bar{l_2}) \\
 & \text{and} & \hat{C}(\bar{l_2}) \simeq_{\downarrow} [\textbf{int}]
\end{array}
$$

5.6 Union-Find

The constraint solving algorithm for Guardol's type system was constructed using well known disjoint-set data structure called Union-Find[9]. A disjoint-set data structure contains no overlapping sets, such that every element in the data structure is unique and belongs to exactly one set. Two operations can be performed on a Union-Find data structure:

- FIND($\overline{var}$) - This operation is used to determine the set to which a particular element belongs.
- UNION($\overline{var}$, $\overline{var}$) - This operation combines or merges two sets into a single set.

The type variable cache ($\hat{C}$), defined in beginning of this chapter, is maintained by our implementation of the Union-Find algorithm. Elements of the data structure are either type variables ($\tilde{\tau}$), expression labels ($\bar{l}$), or variable names ($\bar{x}$). These elements are collectively denoted by $\overline{var}$ in Section 5.1. All of these elements share the ability to be associated with type variables, such that every element in a particular set has the same type. If no type variable exists in a set then that set of elements would not have a defined type, while a set containing two incompatible type variables indicates a typing error.

Elements are introduced into the data structure via the FIND operation. When FIND is called with an element that does not exist in the data structure, it creates a singleton set with that element and returns the inhabiting element. If FIND is called on an existing element within the data structure, an element with the highest rank is returned. Rank will be discussed in depth in the following paragraph, however it is sufficient to know that each set must return an element of its set when FIND is called and in this case rank is used to determine which element is returned. FIND creates a new element every time its argument is a type variable, however if its argument is a variable name or an expression label then it only creates a new element iff an element with that argument does not already exist in the data structure. A new element is created for each type variable because type variables are used to provide information on the type of the set and not unify existing multi-element sets together. Whereas expression labels and variables names are used by the constraint rules to join multi-element sets together. When UNION is called, it performs a FIND on both elements and subsequently merges them into the same set.

It is not strictly necessary to create a new element for each type variable, however this simplifies the approach without adding undo space or time complexity. Coercion, or the unioning, of type names and type structures into the same set is possible but we want to make sure that the type structure is not used to reference the set. For example, if [**int**] is unioned with my_int and in some completely unrelated expression or statement [**int**] is unioned with int, we would not want a type error to occur. A type error, however, would occur if [**int**] could not belong to multiple sets. We solve this problem by assuring that a new element is created when any type variable is introduced, eliminating the possibility that a type variable is used to union sets together.

Rank is a standard mechanism found in many Union-Find algorithms used to manipulate the algorithm's time complexity[9]. Its primary purpose is to balance the tree data structures storing each set such that the asymptotic running time of the FIND operation is faster. For reasons that are clear based on the limiting structure of the AST and the relative number of elements that are created to type each program, we are not necessarily concerned with a faster asymptotic running time. Instead we appropriate the ranking mechanism for a different purpose that allows us to assure that FIND returns an element that provides the highest level of detail regarding the type of the elements in the set. In all cases where FIND is called with an element whose set contains more than one element, FIND will return a type variable. This can be proven inductively over AST on a case by case basis for each type constraint rule.

The algorithm utilizes three different ranks such that the FIND operation returns the element (i.e. type variable, expression label, or variable name) that corresponds with the most precise type description. The lowest rank is comprised of expression labels and variable names which by themselves provide no description of their type. Type variables are split into two separate groups, qualified type names and type structures as defined in Section 4.5. Type structures comprise the middle rank because they provide a description of a type's structure but could be associated with any number of qualified type names which have the same structure. Finally, qualified type names are given the highest rank because they directly reference a user defined or primitive type and are not compatible with any other user defined types.

For example, if we had the variable name i1 and the type variable $[\textbf{int}]^1$ (type variables are uniquely identified by a superscript identifier so that they can be differentiated from other type variables of the same type) a UNION operation would result in these two elements occupying the same set. Any call of the FIND operation on either element would return the type variable $[\textbf{int}]^1$. Later, if the type variable $\mathsf{my\_int}^2$ was unioned to either $[\textbf{int}]^1$ or i1 then a call of the FIND operation with any of these elements would return the type variable my_int^2 because they all occupy the same set and my_int^2 is the element with the highest rank. The "strength" of an element is directly related to its rank. The stronger the element that is returned the more precise the type of that set and the less potential it has to be unioned with other sets. To determine if two elements can be unioned together, we must evaluate whether or not they are equivalent.

The evaluation of equivalence, $\overline{var}_1 \simeq \overline{var}_2$, is split into two cases:

- if $\overline{var_1}$ and $\overline{var_2} \in typeVariable$, $\overline{var}_1$ and $\overline{var}_2$ are equivalent iff the equivalence rules defined in Figure 4.2 hold.
- otherwise, $\overline{var}_1$ and $\overline{var}_2$ are always equivalent.

Algorithmically, $\overline{var}_1 \simeq \overline{var}_2$ is modeled by invoking the UNION operation on $\overline{var}_1$ and $\overline{var}_2$ if they are equivalent. If they are not equivalent, then a type error is raised. This is indicative of a type mismatch where one type was expected by the type system and another type was supplied. If we consider the example in the above paragraph, where we have my_int^2, [**int**]^1, and i1 inhabiting the same set, we would not want a union to occur with any set that contained an element such as int^3 or [**bool**]^4. The resulting set would then have a conflicting type information. A type cannot have two names or be composed of conflicting type structures, such as [**bool**]^4 and [**int**]^1.

We want to guarantee that sets with conflicting type information will never be unioned together. As part of our inductive argument, we will assume that the empty set does not contain conflicting type information and sets whose identifying elements are equivalent do not have conflicting type information. If a set contains int^3 and was not incorrectly unioned previously, when FIND is called on any element in that set we know that some type variable int^n will be returned because those elements have the highest rank. This assures that sets containing different type names cannot be unioned together. If a set contains [**bool**]^4 and was not incorrectly unioned previously, when find is called on any element in that set we know that either some [**bool**]^n or type name such as my_bool will be returned. Since the type name my_bool is stronger than $[\textbf{bool}]^4$ and any other $[\textbf{bool}]^n$ provides identical information, we can be assured that using the evaluation of equivalence before unioning this set with any other will not result in a set with conflicting type information being created.

Equivalence constraints are regarded semantically as operations performed to refine or maintain the type of either the $\overline{var}_1$ argument or the $\overline{var}_2$ argument. Since we can assume that one of the $\overline{var}$s is a type variable, we know that if the equivalence check holds a FIND operation performed on any element in the resulting union set will return a type variable that is as strong or stronger than the type variable that would have been returned before the UNION operation. Informally, this is indicative of each set either maintaining or gaining a more precise definition of its type. The following section provides an example that illustrates the utility of the Union-Find algorithm in conjunction with the type constraint generation rules.

5.7 Type Constraint Generation Example

This type constraint generation example traverses the following assignment statement

```
t1 := external_node(i1+2) when i2;
```

from Figure 4.1. A pair of figures, the AST and the cache ($\hat{C}$), model each of the seven constraint generation steps required to type check this statement. For the sake of this example, we will assume that the nested type constraint rule calls, e.g. $\langle \hat{C},\ \Sigma,\ \Psi \rangle \models_E \ \bar{e}^{\bar{l}}$, are visited in the order in which they are declared within the type constraint rules. This will cause the AST to be traversed depth first from left to right. The order that the child nodes are visited, however, is not important algorithmically or semantically as long as the AST is traversed depth first. Each expression node in the AST is given a unique expression label that will eventually be mapped to a type variable. This mapping is represented by $\hat{C}$.

Though $\hat{C}$ is algorithmically modeled using a Union-Find data structure as explained in Section 5.6, we will model it visually as a table with variable names and expression labels on the left and the corresponding type variable on the right. A superscript number is used to uniquely identify the type variable representing a disjoint set of elements. For the purpose of being able to identify different type variables of the same type, a unique identifier is used for type variables. So if i1 and l_3 are both identified by $\mathsf{my\_int}^1$, then they are all in the same set. However, if one is identified by $\mathsf{my\_int}^1$ and the other is identified by $\mathsf{my\_int}^2$ they are in different sets.

From the perspective of the Union-Find algorithm, the type variable in the right cell is found by calling FIND on the variable name or expression label (e.g. $\hat{C}(\mathsf{i1})$ returns $\mathsf{my\_int}^1$). An empty cell signifies that FIND returned a variable name or expression label that was identical to its argument, meaning that the variable name or expression label is the only inhabitant its disjoint set.

5.7.1 Step 1

We begin by processing declarations in the following code which results in initial constraints for $\hat{C}$ being processed as illustrated in Figure 5.4.

```
node internal_node
  (i1 : my_int, i2 : my_bool) returns
  (o1 : bool)
is
local
  t1 : my_tuple;
```

The variable name declaration AST nodes are visited and the corresponding Type Constraint Rule 5.1 is evaluated. The variables i1, i2, t1, and o1 have been unioned with their respective type variables: $\mathsf{my\_int}^1$, $\mathsf{my\_bool}^2$, $\mathsf{my\_tuple}^3$, and bool^4.

In Step 1, we will generate the constraints for the node $\mathsf{i1}^{l_4}$ in Figure 5.3. This node matches Type Constraint Rule 5.19 and the constraint $\hat{C}(l_4) \simeq_{\downarrow} \hat{C}(\mathsf{i1})$ is generated. Where $\hat{C}(\mathsf{i1})$ returns $\mathsf{my\_int}^1$ and $\hat{C}(l_4)$ returns l_4. After the union operation, we see that the expression label l_4 now has the type $\mathsf{my\_int}^1$ in Figure 5.4.

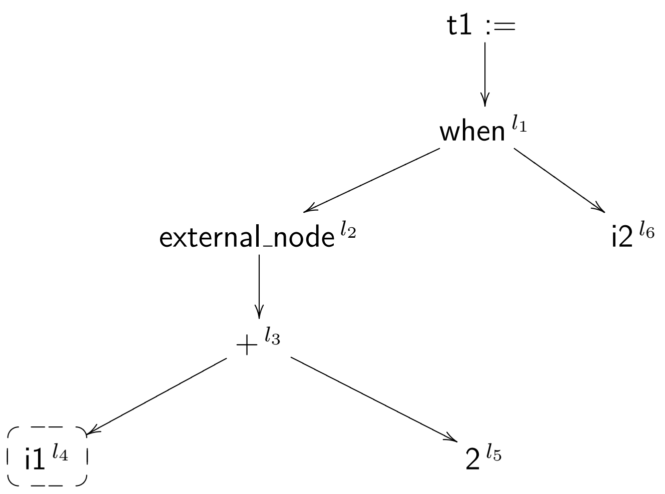

Figure 5.3: AST - Step 1

i1	my_int^1
i2	my_bool^2
t1	my_tuple^3
o1	bool^4
l_1	
l_2	
l_3	
l_4	my_int^1
l_5	
l_6	

Figure 5.4: Cache ($\hat{C}$) - Step 1

5.7.2 Step 2

In Step 2, we will generate the constraints for the node 2^{l_5} in Figure 5.5. This node matches Type Constraint Rule 5.21 and the constraint $\hat{C}(\bar{l_5}) \simeq_{\downarrow}$ [**int**] is generated. In Figure 5.6 we can see the expression label l_5 is now unioned with the primitive type structure [**int**]^5. This type structure will eventually be strengthened by its replacement with a qualified type name, however for the type constraint to be satisfied the structure of the type name must be equivalent to [**int**].

5.7.3 Step 3

In Step 3, we will generate the constraints for the node $+^{l_3}$ in Figure 5.7. This node matches Type Constraint Rule 5.40 and the constraints $\hat{C}(l_3) \simeq_{\downarrow} \hat{C}(l_4)$, $\hat{C}(l_4) \simeq_{\downarrow} \hat{C}(l_5)$, $\hat{C}(l_5) \simeq_{\downarrow}$ [**int**]. Where $\hat{C}(l_4)$ returns my_int^1, $\hat{C}(l_5)$ returns [**int**]^5, and $\hat{C}(l_3)$ returns l_3. We must union my_int^1, [**int**]^5, and l_3. There is no type constraint violation since the primitive structural type, [**int**]^5, is type equivalent to my_int^1. In Figure 5.8, note that $\hat{C}(l_5)$ has changed from [**int**]^5 to my_int^1 due the nature of our Union-Find algorithm as explained in Section 5.6. We also know that collectively these constraints assure that the arguments of the operator + are type equivalent to [**int**] as well as the resulting expression.

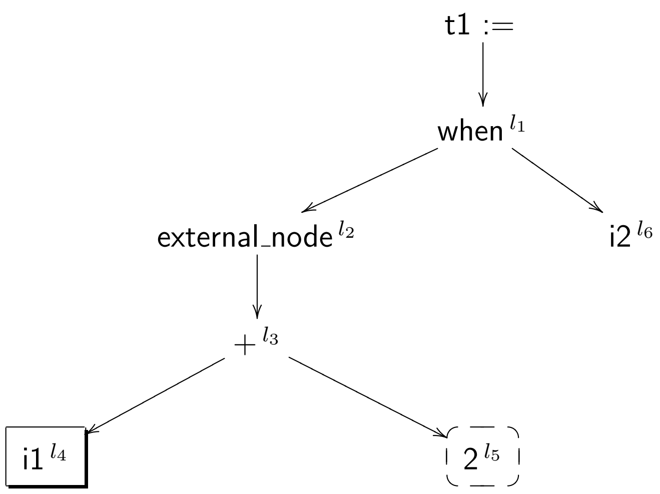

Figure 5.5: AST - Step 2

i1	my_int^1
i2	my_bool^2
t1	my_tuple^3
o1	bool^4
l_1	
l_2	
l_3	
l_4	my_int^1
l_5	[**int**]^5
l_6	

Figure 5.6: Cache ($\hat{C}$) - Step 2

Algorithmically, we ensure this constraint by always unioning one of the expression labels, $\hat{C}(l_5)$, with [**int**]. This may cause multiple structural checks if the arguments are numeric literals or expressions that have primitive types, such as 2 or $6 * 9$.

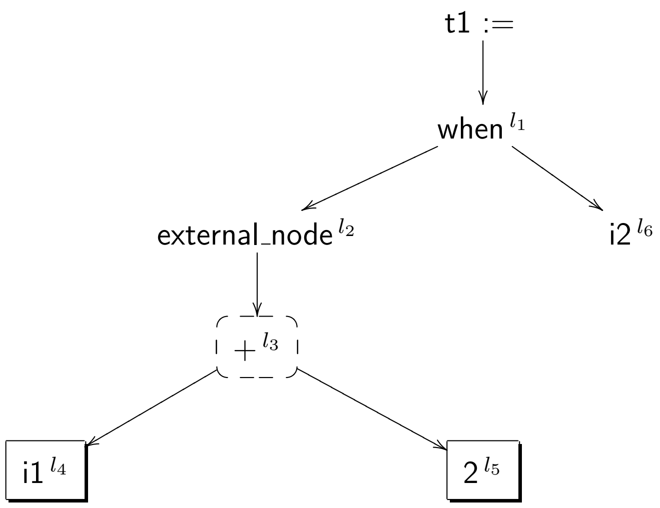

Figure 5.7: AST - Step 3

i1	my_int^1
i2	my_bool^2
t1	my_tuple^3
o1	bool^4
l_1	
l_2	
l_3	my_int^1
l_4	my_int^1
l_5	my_int^1
l_6	

Figure 5.8: Cache ($\hat{C}$) - Step 3

5.7.4 Step 4

In Step 4, we will generate the constraints for the node $\mathsf{external\_node}^{l_2}$ in Figure 5.9. This node matches Type Constraint Rule 5.18 and the constraints $\hat{C}(l_3) \simeq_\downarrow \Psi_{in}(\mathsf{external\_node}, 1)$ and $\hat{C}(l_2) \simeq_\downarrow [\Psi_{out}(\mathsf{external\_node}, 1) * \Psi_{out}(\mathsf{external\_node}, 2)]$ are generated. Using the node parameter lookup function we find that $\Psi_{in}(\mathsf{external\_node}, 1)$ returns $\mathsf{my\_int}^7$, $[\Psi_{out}(\mathsf{external\_node}, 1)*\Psi_{out}(\mathsf{external\_node,2})]$ returns $[\mathsf{int\ *\ bool}]^6$, $\hat{C}(l_3)$ returns $\mathsf{my\_int}^1$, and $\hat{C}(l_2)$ returns l_2. First, we must union $\mathsf{my\_int}^1$ and $\mathsf{my\_int}^7$ which are equivalent. Next, in Figure 5.10, the expression label l_2 is unioned with $[\mathsf{int\ *\ bool}]^6$.

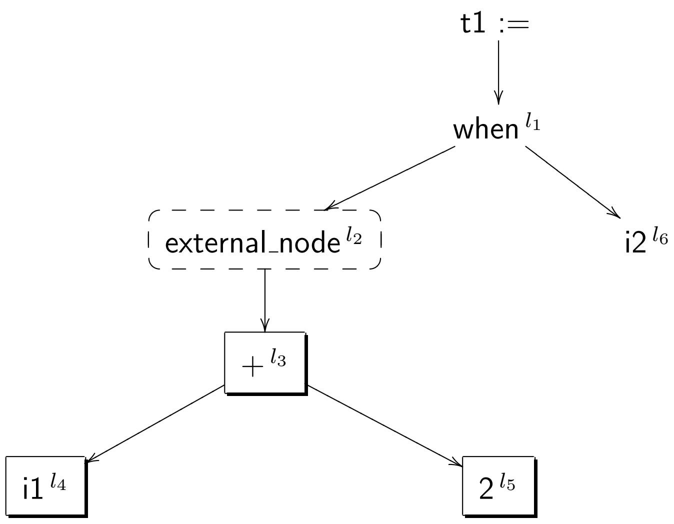

Figure 5.9: AST - Step 4

i1	$\mathsf{my\_int}^1$
i2	$\mathsf{my\_bool}^2$
t1	$\mathsf{my\_tuple}^3$
o1	bool^4
l_1	
l_2	$[\mathsf{int\ *\ bool}]^6$
l_3	$\mathsf{my\_int}^1$
l_4	$\mathsf{my\_int}^1$
l_5	$\mathsf{my\_int}^1$
l_6	

Figure 5.10: Cache ($\hat{C}$) - Step 4

5.7.5 Step 5

In Step 5, similar to Step 1, we will generate the constraints for the node $\mathsf{i2}^{l_6}$ in Figure 5.11. This node matches Type Constraint Rule 5.19 and the constraint $\hat{C}(l_6) \simeq_\downarrow \hat{C}(\mathsf{i2})$ is generated. Where $\hat{C}(\mathsf{i2})$ returns $\mathsf{my\_bool}^2$ and $\hat{C}(l_6)$ returns l_6. In Figure 5.12, the expression label l_6 is unioned with $\mathsf{my\_bool}^2$.

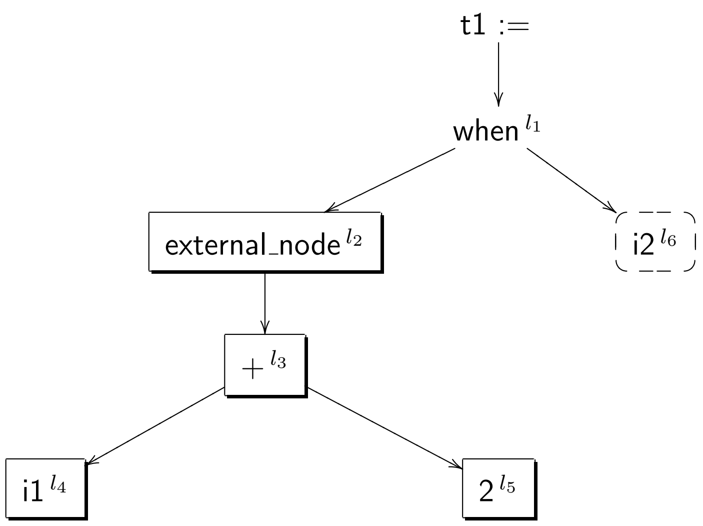

Figure 5.11: AST - Step 5

i1	my_int^1
i2	my_bool^2
t1	my_tuple^3
o1	bool^4
l_1	
l_2	[int * bool]^6
l_3	my_int^1
l_4	my_int^1
l_5	my_int^1
l_6	my_bool^2

Figure 5.12: Cache ($\hat{C}$) - Step 5

5.7.6 Step 6

In Step 6, we will generate the constraints for the node when^{l_1} in Figure 5.13. This node matches the Type Constraint Rule 5.32 and the constraints $\hat{C}(l_6) \simeq_{\downarrow}$ [**bool**] and $\hat{C}(l_1) \simeq_{\downarrow} \hat{C}(l_2)$ are generated. Where $\hat{C}(l_6)$ returns my_bool^2, $\hat{C}(l_2)$ returns [int * bool]^6, and $\hat{C}(l_1)$ returns l_1. The first constraint ensures that my_bool^2 is type equivalent to [**bool**] which allows us to verify that the second argument of a when expression is typed correctly. In Figure 5.14, expression label l_1 has is assigned with [int * bool]^6 and the type of the when expression is equivalent to [int * bool]^6.

5.7.7 Step 7

In Step 7, we will generate the constraints for the node t1 := in Figure 5.15. This node matches the Type Constraint Rule 5.3 and the constraint $\hat{C}(l_1) \simeq_{\downarrow} \hat{C}$(t1) is generated. Where $\hat{C}(l_1)$ returns [int * bool]^6 and $\hat{C}$(t1) returns my_tuple^3. Referring to the type declaration of my_tuple

```
type my_tuple is int * bool;
```

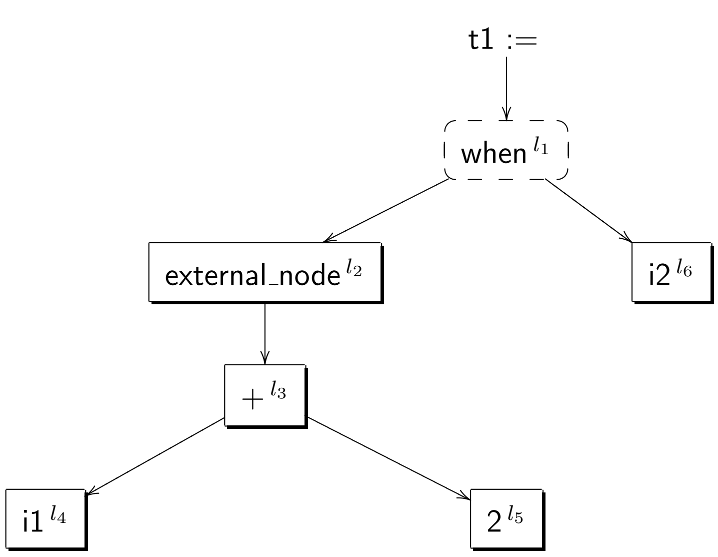

Figure 5.13: AST - Step 6

i1	my_int^1
i2	my_bool^2
t1	my_tuple^3
o1	bool^4
l_1	▨ [int * bool]^6
l_2	▨ [int * bool]^6
l_3	my_int^1
l_4	my_int^1
l_5	my_int^1
l_6	▨ my_bool^2

Figure 5.14: Cache ($\hat{C}$) - Step 6

in Figure 4.1, we can see that this constraint is satisfied because [int * bool]^6 and my_tuple^3 are equivalent. The union of these two type variables also results in a strengthening of the type [int * bool]^6 to my_tuple^3 in the cache; the resulting type variables occurring from $\hat{C}(()l_1)$ and $\hat{C}(()l_2)$ are adjusted to standardize this strengthening. Algorithmically, this process is ensured by the simple union of the two type variables, where the type names (i.e. my_tuple^3) will always have a higher rank than type structures (i.e. [int * bool]^6). This node call whose type structure is [int * bool]^6 has been strengthened to the type name my_tuple^3. This does not affect the type of any subsequent node calls or the referenced node's parameter type definition.

Type constraint solving fails when any constraint fails. If a constraint cannot be unioned, then a type mismatch error is raised for that constraint relating the expression labels and/or variable names with their expected/given types. Failure to adhere to predicate constraint determinations are also related in error messages. A match statement that expected a constructor with a specific name and was given another is an example of a predicate constraint failure. Error messages are detailed with line numbers and provide as much of the expected/given type information about the violating expression labels and variable names as

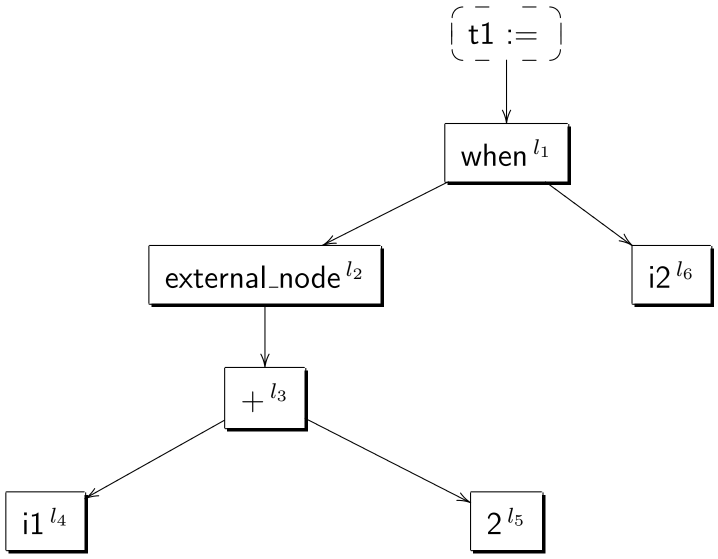

Figure 5.15: AST - Step 7

i1	my_int^1
i2	my_bool^2
t1	my_tuple^3
o1	bool^4
l_1	my_tuple^3
l_2	my_tuple^3
l_3	my_int^1
l_4	my_int^1
l_5	my_int^1
l_6	my_bool^2

Figure 5.16: Cache ($\hat{C}$) - Step 7

possible. For ease of use, the constraint solving algorithm does not stop after one constraint fails. It caches the error and continues without the evaluation or union resulting from the violating constraint.

Chapter 6

Future Work

The Guardol compiler composed of the grammar, parser, AST, translation to the Pilar IR, and type system represents full scope of the implementation provided by this report. The purpose of this project, as explained in Section 1.4, was to provide a foundation upon which various verification analysis, target platform translations, and external tools could be harnessed. The scope of future work includes the implementation and formalization of these tool chains to provide a high degree of confidence in the resulting product.

6.1 Specification Language

A large part of future development efforts will center around the creation and use of a guard specification language. This language is a critical component of the guard design process, where straightforward abstractions can be understood from a natural language perspective. The necessity of guarding applications often stems from simple written directives describing an information flow policy that must be enforced. A specification language that provides the appropriate abstractions, in an unambiguous manner, corresponding directly to the style in which guarding policies are conceptualized and formally written.

While, informally, the is the goal of most DSLs is to provide a framework for computing in a given domain, this specification language's purpose would be to facilitate the construction of guarding policies that are readable, unambiguous, and applicable to the Guardol tool-chain. The specification of guarding policies is the first step in the design and implementation of a guarding solution. As such, the specification language should be able to be compiled at any point in the design process and checked to verify that it is well-formed with respect to the specification language. The compilation of a specification, however, will not yield an implementation.

To verify properties about the implementation of a guarding solution some sort of specification must exist, so it is likely that implementations will not be able to be compiled without a specification. While the specification and the implementation languages are compiled separately, it is not unreasonable to assume that developers would want the shared abstractions (e.g. data types, node definitions) in the specification and implementation languages to inhabit the same file or set of files. A comprehensive grammar that encapsulates both languages such that the two languages are merged but maintain separate compiling phases is a feasible development goal.

6.2 Pilar Intermediate Representation

The Pilar IR was designed by the SAnToS group so static analysis such as information flow and symbolic execution can be applied to all Pilar IR translations conforming to the corresponding language profiles. Guardol was designed as a single assignment language with an added restriction requiring that referenced variables could not subsequently be assigned any value. Regardless of this restriction to the single assignment paradigm, we know that a single assignment language is a functional language[10]. Therefore a generic functional language profile for the Pilar IR would be an acceptable target for many of Guardol's static analysis tools.

Currently, there is no documentation or formalization for the Pilar language profiles as it is an implementation mechanic that is under development. However, providing existing static analysis tools that work directly on the Pilar IR is advantageous during the development of DSLs which do not differ greatly from standard language paradigms. Static analysis tools developed specifically for the DSL must be re-certified or proven for that implementation. In contrast, a DSL that uses existing static analysis tools through the Pilar IR requires that the translation from the DSL to the IR be certified or proven correct. This decision is a choice of implementation strategy and not necessarily correctness. It is conceivable, however, that one implementation strategy provide a framework that is significantly easier to certify and/or prove.

6.3 Verification

The verification of guard implementations on target platforms adhering to a given specification is the end to end goal of the Guardol project. Currently, a majority of the focus has been targeted at the integration of static analysis tools to verify a guard's functional equivalence to the provided specification. Dynamic analysis, such as runtime checks, are not the type of analysis tools that we are interested in implementing to prove that guard implementation adheres to a specification or guard policy. Relying on this type of analysis would provide no guarantees about the guards behavior at runtime and if it was used to enforce behavior then it should be used in place of the guard as a verified external tool.

The process of verification will rely heavily on future decisions regarding the construction of the specification language. If further efforts are made to design a language that maintains property specifications on the inputs and outputs of nodes (e.g. VirusChecked, Scrubbed), a framework for reasoning about these properties and possibly in a temporal context would be the next step. A readable and intuitive specification of properties and when they hold is critical to the design of the language as well as the verification process.

Verification on different target platforms requires that the translations from the Pilar IR to each platform be certified or that properties of the specification are maintained at every step in the tool-chain. This by-product of platform independence will be the subject of a great deal of future effort. Many different and possibly redundant strategies must be used to accomplish this task in such a way that we have a "high assurance" that the resulting implementation is correct.

Bibliography

[1] Chairman Of The Joint Chiefs Of Staff Instruction 6211.02B, CJCSI Distribution July, 2003.

[2] J. P. Anderson, Computer Security Technology Planning Study, Technical Report ESD-TR-73-51, US Air Force, 1972.

[3] Trusted Network Environment (TNE), General Dynamics - Advanced Information Systems, 2006.

[4] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*, Pragmatic Programmers, Pragmatic Bookshelf, first edition, 2007.

[5] E. Gamma, *Design Patterns*, Addison-Wesley, Boston, 1995.

[6] C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, John Wiley & Sons, 2002.

[7] J. Gao, M. Whalen, and E. Van Wyk, Electron. Notes Theor. Comput. Sci. **203**, 111 (2008).

[8] B. C. Pierce, *Types and Programming Languages*, The MIT Press, 2002.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, 2001.

[10] A. W. Appel, SSA is Functional Programming, 1998.

Appendix A

Grammar

This appendix provides the grammar for the syntax of Guardol. There is a direct correspondence between these grammar production and the ANTLR rules used in the Guardol parser.

A.1 Names

$$\textit{name} \quad ::= \quad \textbf{IDENTIFIER} \tag{A.1}$$

$$\textit{packageName} \quad ::= \quad \textit{name} \tag{A.2}$$

$$\textit{nodeName} \quad ::= \quad \textit{name} \tag{A.3}$$

$$\textit{typeName} \quad ::= \quad \textit{name} \tag{A.4}$$

$$\textit{constructorName} \quad ::= \quad \textit{name} \tag{A.5}$$

$$\textit{variableName} \quad ::= \quad \textit{name} \tag{A.6}$$

$$\textit{constName} \quad ::= \quad \textit{name} \tag{A.7}$$

$$\mathit{fieldName} ::= \mathit{name} \tag{A.8}$$

$$\begin{aligned} \mathit{qualifiedPackageName} &::= \mathit{packageName} \\ &\mid \mathit{packageName}\ \mathbf{.}\ \mathit{qualifiedPackageName} \end{aligned} \tag{A.9}$$

$$\begin{aligned} \mathit{qualifiedNodeName} &::= \mathit{nodeName} \\ &\mid \mathit{packageName}\ \mathbf{.}\ \mathit{qualifiedNodeName} \end{aligned} \tag{A.10}$$

$$\begin{aligned} \mathit{qualifiedTypeName} &::= \mathit{typeName} \\ &\mid \mathit{packageName}\ \mathbf{.}\ \mathit{qualifiedTypeName} \end{aligned} \tag{A.11}$$

$$\begin{aligned} \mathit{componentTypeName} &::= \mathit{primitiveType} \\ &\mid \mathit{qualifiedTypeName} \end{aligned} \tag{A.12}$$

A.2 Type Declarations

$$\mathit{typeDeclaration} ::= \mathbf{type}\ \mathit{typeName}\ \mathbf{is}\ \mathit{typeDefinition}\mathbf{;} \tag{A.13}$$

$$\mathit{andTypeDeclaration} ::= \mathbf{and}\ \mathit{typeName}\ \mathbf{is}\ \mathit{typeDefinition}\mathbf{;} \tag{A.14}$$

$$\begin{aligned} \mathit{typeDefinition} &::= \mathit{unionType} \\ &\mid \mathit{recordType} \\ &\mid \mathit{listType} \\ &\mid \mathit{simpleType} \\ &\mid \mathit{componentType} \\ &\mid \mathbf{external} \end{aligned} \tag{A.15}$$

$$\mathit{primitiveType} ::= \mathbf{bool} \tag{A.16}$$

	$\mid$	**int**	
	$\mid$	**real**	
	$\mid$	**char**	
	$\mid$	**string**	
unionType	::=	**union** *variants* **end union**	(A.17)
variants	::=	*variant*	(A.18)
	$\mid$	*variant* $\mid$ *variants*	
variant	::=	*constructorName*	(A.19)
	$\mid$	*constructorName* **of** *componentType*	
recordType	::=	**record** *fieldDeclarations* **end record**	(A.20)
fieldDeclarations	::=	*fieldDeclaration*	(A.21)
	$\mid$	*fieldDeclaration* **,** *fieldDeclarations*	
fieldDeclaration	::=	*fieldName* **:** *componentType*	(A.22)
listType	::=	**list** *componentType* **end list**	(A.23)
simpleType	::=	*componentTypeName*	(A.24)
tupleType	::=	*componentTypeName* ***** *componentTypeName*	(A.25)
	$\mid$	*componentTypeName* ***** *tupleType*	
componentType	::=	*simpleType*	(A.26)
	$\mid$	*tupleType*	

A.3 Constant Declarations

constantDeclaration	::=	**constant** *constName* **:** *componentType constAssign*	(A.27)
constAssign	::=	**:=** *literalExpression*	(A.28)
	|	**is external**	

A.4 Expressions

expressions	::=	*expression*	(A.29)
	|	*expression* **,** *expressions*	
expression	::=	*whenExpression*	(A.30)
nameExpression	::=	*variableName*	(A.31)
literalExpression	::=	**“ STRING “**	(A.32)
	|	**’ CHARACTER ’**	
	|	**true**	
	|	**false**	
	|	**INTEGER**	
	|	**REAL_NUMBER**	
parenExpression	::=	**(** *expression* **)**	(A.33)
callExpression	::=	*qualifiedNodeName* **(** *expressions* **)**	(A.34)
	|	*qualifiedNodeName* **()**	

simpleExpression	::=	*nameExpression*	(A.35)
	|	*literalExpression*	
	|	*parenExpression*	
	|	*callExpression*	
tupleAccess	::=	*simpleExpression* ♯ **NATURAL_NUMBER**	(A.36)
recordAccess	::=	*simpleExpression* **.** *fieldName*	(A.37)
unionConstruction	::=	*typeName* **'** *constructorName*	(A.38)
	|	*typeName* **'** *constructorName* **(** *expression* **)**	
tupleConstruction	::=	*typeName* **'(** *fieldConstructions* **)**	(A.39)
recordConstruction	::=	*typeName* **'[** *fieldConstructions* **]**	(A.40)
fieldConstructions	::=	*fieldConstruction*	(A.41)
	|	*fieldConstruction* **,** *fieldConstructions*	
fieldConstruction	::=	*fieldName* **=>** *expression*	(A.42)
listConstruction	::=	*typeName* **'{** *expressions* **}**	(A.43)
	|	*typeName* **'nil**	
	|	*typeName* **'{}**	
unaryBooleanOp	::=	**not**	(A.44)
unaryNumericOp	::=	**+**	(A.45)
	|	**-**	

unaryExpression	::=	*unaryBooleanOp expression*	
	|	*unaryNumericOp expression*	(A.46)
existsExpression	::=	**exists** *expression*	(A.47)
compExpression	::=	*simpleExpression*	(A.48)
	|	*tupleAccess*	
	|	*recordAccess*	
	|	*tupleConstruction*	
	|	*recordConstruction*	
	|	*listConstruction*	
	|	*unionConstruction*	
	|	*unaryExpression*	
	|	*existsExpression*	
binaryNumericOp1	::=	/	(A.49)
	|	%	
	|	*	
binaryNumericExpression1	::=	*compExpression*	(A.50)
	|	*compExpression binaryNumericOp1*	
		binaryNumericExpression1	
binaryNumericOp2	::=	+	(A.51)
	|	-	
binaryNumericExpression2	::=	*binaryNumericExpression1*	(A.52)

	|	*binaryNumericExpression1 binaryNumericOp2*	
		binaryNumericExpression2	
binaryRelationalOp	::=	=	(A.53)
	|	/=	
	|	< =	
	|	> =	
	|	<	
	|	>	
binaryRelationalExpression	::=	*binaryNumericExpression2*	(A.54)
	|	*binaryNumbericExpression2 binaryRelationalOp*	
		binaryRelationalExpression	
binaryBooleanOp	::=	**and**	(A.55)
	|	**or**	
binaryBooleanExpression	::=	*binaryRelationalExpression*	(A.56)
	|	*binaryRelationalExpression binaryBooleanOp*	
		binaryBooleanExpression	
consExpression	::=	*binaryBooleanExpression*	(A.57)
	|	*binaryBooleanExpression* **::** *consExpression*	
appendExpression	::=	*consExpression*	(A.58)
	|	*consExpression* **@** *appendExpression*	

defaultExpression	::=	*appendExpression*	(A.59)
	|	*appendExpression* **default** *defaultExpression*	
whenExpression	::=	*defaultExpression*	(A.60)
	|	*defaultExpression* **when** *whenExpression*	

A.5 Statements

statementList	::=	*statement* **;**	(A.61)
	|	*statement* **;** *statementList*	
statement	::=	*assignmentStatement*	(A.62)
	|	*matchStatement*	
	|	*ifStatement*	
	|	*skipStatement*	
assignmentStatement	::=	*varIdentifiers* **:=** *expression*	(A.63)
matchStatement	::=	**match** *expression* **with** *clauses* **end match**	(A.64)
ifStatement	::=	**if** *expression* **then** *statementList* *elseStatement*	(A.65)
elseStatement	::=	**elseif** *expression* **then** *statementList* *elseStatement*	(A.66)
	|	**else** *statementList* **end if ;**	
	|	**end if ;**	

skipStatement	::=	**skip**	(A.67)
varIdentifiers	::=	*variableName*	(A.68)
	|	*variableName* **,** *varIdentifiers*	
clauses	::=	*clause*	(A.69)
	|	*clause* | *clauses*	
clause	::=	*pattern* **=>** *statementList*	(A.70)
pattern	::=	*variableName*	(A.71)
	|	*consList*	
	|	**'***constructorName*	
	|	**'***constructorName variableName*	
consList	::=	**nil**	(A.72)
	|	*variableName* **::** *variableName*	

A.6 Node Declarations

nodeDeclaration	::=	**node** *nodeName*	(A.73)
		nodeParams **returns** *nodeParams*	
		is *nodeBody* **;**	
nodeBody	::=	**external**	(A.74)
	|	**begin** *statementList* **end node**	
	|	**local** *nameDeclarations* **begin** *statementList* **end node**	

nodeParams	::=	( )	(A.75)
	|	(*nameDeclarations*)	
nameDeclarations	::=	*nameDeclaration*	(A.76)
	|	*nameDeclaration* , *nameDeclarations*	
partialType	::=	*componentType*	(A.77)
	|	*componentType* **total**	
nameDeclaration	::=	*variableName* **:** *partialType*	(A.78)

A.7 Package Declarations

packageDeclarations	::=	*packageDeclaration packageDeclarations*	(A.79)
	|	*empty*	
packageDeclaration	::=	*withClauses*	(A.80)
		package *packageName* **is**	
		typeDeclarations constantDeclarations	
		nodeDeclarations	
		end package ;	
typeDeclarations	::=	*typeDeclaration* ; *typeDeclarations*	(A.81)
	|	*typeDeclaration* ; *andTypeDeclarations*	
	|	*empty*	
andTypeDeclarations	::=	*andTypeDeclaration* ; *typeDeclarations*	(A.82)

$$
\begin{array}{llll}
 & \mid & \mathit{andTypeDeclaration}\ ;\ \mathit{andTypeDeclarations} & \\
\mathit{constantDeclarations} & ::= & \mathit{constantDeclaration}\ ;\ \mathit{constantDeclarations} & \text{(A.83)} \\
 & \mid & \mathit{empty} & \\
\mathit{nodeDeclarations} & ::= & \mathit{nodeDeclaration}\ ;\ \mathit{nodeDeclarations} & \text{(A.84)} \\
 & \mid & \mathit{empty} & \\
\mathit{withClauses} & ::= & \mathit{withClause}\ \mathit{withClauses} & \text{(A.85)} \\
 & \mid & \mathit{empty} & \\
\mathit{withClause} & ::= & \textbf{with}\ \mathit{packageIdentifiers}\ ; & \text{(A.86)} \\
\mathit{packageIdentifiers} & ::= & \mathit{qualifiedPackageName} & \text{(A.87)} \\
 & \mid & \mathit{qualifiedPackageName}\ ,\ \mathit{qualifiedPackageName} &
\end{array}
$$

Appendix B

ANTLR Grammar

```
/*******************************************************************************
 * Copyright (c) 2009 Jonathan Hoag, Kansas State University, and others.      *
 * All rights reserved. This program and the accompanying materials            *
 * are made available under the terms of the Eclipse Public License v1.0       *
 * which accompanies this distribution, and is available at                    *
 * http://www.eclipse.org/legal/epl-v10.html                                   *
 *                                                                             *
 * Contributors:                                                               *
 *     Jonathan Hoag - initial API and implementation                          *
 ******************************************************************************/

grammar guardol;

options { backtrack=true; memoize=true; }

@header {

        package org.sireum.profile.guardol.parser;

        import org.sireum.base.message.Message;
        import org.sireum.base.message.ErrorMessage;

        import java.util.ArrayList;

        import org.sireum.profile.guardol.model.*;

        import org.sireum.profile.guardol.selection.RegionSelection;
        import org.sireum.profile.guardol.selection.Caret;


        /**
         * Guardol parser.
         *
         * @author jonathan hoag
         */
}

@lexer::header {
        package org.sireum.profile.guardol.parser;

        /**
         * Guardol lexer.
         *
         * @author jonathan hoag
         */
}

@members {
        protected ArrayList<Message> errors = new ArrayList<Message>();

        private String source = null;

        public List<Message> popErrors() {
          List<Message> result = errors;
          errors = new ArrayList<Message>();
          return result;
        }

        public void setSource(String s) {
          source = s;
        }
        @Override
        public void displayRecognitionError(String[] tokenNames,
          RecognitionException re) {
          final ErrorMessage em = new ErrorMessage();
          em.setTheLineNumber(re.line);
          final String s = re.token.getText();
          final String[] ss = s.split("\n");
          final int len = ss.length;
          em.setTheColumnNumber(re.charPositionInLine + 1);
          em.setTheOptionalEndLineNumber(re.line + len - 1);
          em.setTheOptionalEndColumnNumber(re.charPositionInLine +
                        ss[len - 1].length());
          em.setTheMessageText(getErrorMessage(re, tokenNames));
          em.setTheOptionalSource(this.source);
          this.errors.add(em);
        }
}


compilation returns [ Compilation result = null ]
        : c=compilationFile EOF             { result = $c.result; }
        ;

compilationFile returns [ Compilation result = new Compilation() ]
@init{ArrayList<PackageDeclaration> packages =
                new ArrayList<PackageDeclaration>(); }
        : (p=packageDeclaration                          {packages.add($p.result);}
          )*                                     {result.setThePackageDeclarations(packages);}
        ;

withClause returns
                [ ArrayList<PackageName> result = new ArrayList<PackageName>() ]
        : 'with' n=packageIdentifier                     {result.add($n.result);}
        (',' n=packageIdentifier                         {result.add($n.result);}
        )* ';'
        ;

packageDeclaration returns
                [ PackageDeclaration result = new PackageDeclaration() ]
@init{  RegionSelection selection = new RegionSelection(false);}
        : (w=withClause                 {result.setTheWithPackageNames($w.result);}
                                {selection.setStart($w.result.get(0).getTheSelection().getStart());}
           )?
        p='package' i=packageIdentifier 'is'
                        {if(selection.isEmpty())selection.setStart(new Caret($p.line,$p.pos));}
                                        {result.setThePackageName($i.result);}
        t=typeDeclarations                      {result.setTheTypeDeclarations($t.result);}
        c=constantDeclarations                  {result.setTheConstantDeclarations($c.result);}
        n=nodeDeclarations                      {result.setTheNodeDeclarations($n.result);}
        'end' 'package' l=';'                   {selection.setEnd(new Caret($l.line, $l.pos+1));
                                         result.setTheSelection(selection);}
        ;
typeDeclarations returns
                [ ArrayList<TypeDeclaration> result = new ArrayList<TypeDeclaration>() ]
@init{ ArrayList<ID> currentAndTypeIDs = new ArrayList<ID>();
        ArrayList<TypeDeclaration> currentAndTypes = new ArrayList<TypeDeclaration>();}
        : (t=singleTypeDeclaration                       {result.add($t.result);
                                         currentAndTypes.add($t.result);
                                         currentAndTypeIDs.add($t.result.getTheID());}
           ( t=singleTypeDeclaration                     {result.add($t.result);
                                         for(TypeDeclaration type : currentAndTypes){
                                           type.getTheAndTypeIDs().addAll(currentAndTypeIDs);
                                           type.getTheAndTypeIDs().remove(type.getTheID());
                                         }
                                         currentAndTypeIDs = new ArrayList<ID>();
                                         currentAndTypes = new ArrayList<TypeDeclaration>();
                                         currentAndTypes.add($t.result);
                                         currentAndTypeIDs.add($t.result.getTheID());}
           | t=andTypeDeclaration                        {result.add($t.result);
                                         currentAndTypes.add($t.result);
                                         currentAndTypeIDs.add($t.result.getTheID());}
           )*
          )?                            {for(TypeDeclaration type : currentAndTypes){
                                           type.getTheAndTypeIDs().addAll(currentAndTypeIDs);
                                           type.getTheAndTypeIDs().remove(type.getTheID());
                                          }
                                         }
        ;

singleTypeDeclaration returns [TypeDeclaration result = null ]
        : l='type' t=typeDeclaration                     {result = $t.result;}
                                        {result.getTheSelection().setStart(new Caret($l.line,$l.pos));}
        ;

andTypeDeclaration returns [TypeDeclaration result = null ]
        : l='and' t=typeDeclaration                      {result = $t.result;}
                                        {result.getTheSelection().setStart(new Caret($l.line,$l.pos));}
        ;
```

```
typeDeclaration returns [ TypeDeclaration result = null ]
@init{  RegionSelection selection = new RegionSelection(false);}
        : i=identifier
          ( 'is'
            ( 'external'                        {result = new AbstractTypeDeclaration();}
            | u=unionType                       {result = $u.result;}
            | r=recordType                      {result = $r.result;}
            | li=listType                       {result = $li.result;}
            | s=simpleType                      { SimpleTypeDeclaration st = new SimpleTypeDeclaration();
                                          st.setTheSimpleType((SimpleType)$s.result);
                                          result = st;
                                        }
            | t=tupleType                       {TupleTypeDeclaration tt = new TupleTypeDeclaration();
                                          tt.setTheTupleType((TupleType)$t.result);
                                          result = tt;
                                        }
            )
          )
          l=';'                         {result.setTheID($i.text);}
                                        {selection.setEnd(new Caret($l.line, $l.pos+1));
                                         result.setTheSelection(selection);}
        ;


componentType returns[ Type result = null]
        : p=partialType                 {result = $p.result;}
        | t =tupleType                  {result = $t.result;}
        ;

partialType returns [SimpleType result = null]
        : s=simpleType                  {result = $s.result;}
          ('total'                              {result.setTheQualifier(Qualifier.TOTAL);}
          )?
        ;

simpleType returns [ SimpleType result = new SimpleType()]
        : p= pathIdentifier                     {result.setTheTypeID($p.result);
                                         result.setTheOptionalPackageName($p.parent);
                                         result.setTheSelection($p.result.getTheSelection());}
        ;

tupleType returns [ TupleType result = new TupleType()]
        : s= partialType '*'                    {result.setTheSelection(
                                          new RegionSelection(
                                             $s.result.getTheSelection().getStart(),
                                             $s.result.getTheSelection().getEnd(),false));}
                                        {result.getTheSimpleTypes().add($s.result);}
          s= partialType                        {result.getTheSimpleTypes().add($s.result);}
          ( '*' s=partialType                   {result.getTheSimpleTypes().add($s.result);}
                        {result.getTheSelection().setEnd($s.result.getTheSelection().getEnd());}
          )*
        ;

listType returns [ListTypeDeclaration result = new ListTypeDeclaration()]
        :  'list' c=componentType                       {result.setTheType($c.result);}
           'end' 'list'
        ;


unionTypeElement returns [UnionTypeElement result = new UnionTypeElement()]
        : i=identifier                  {result.setTheUnionID($i.text);}
                                        {result.setTheSelection(
                                           new RegionSelection(
                                             $i.text.getTheSelection().getStart(),
                                             $i.text.getTheSelection().getEnd(),false));}
          ('of' t=componentType                 {result.setTheOptionalType($t.result);}
                        {result.getTheSelection().setEnd($t.result.getTheSelection().getEnd());}
          )?
        ;

unionType returns [ UnionTypeDeclaration result = new UnionTypeDeclaration() ]
@init{  ArrayList<UnionTypeElement> elems = new ArrayList<UnionTypeElement>();}
        : 'union'
         e=unionTypeElement                     {elems.add($e.result);}
          ('|' e=unionTypeElement                       {elems.add($e.result);}
          )*
          'end' 'union'                         {result.setTheUnionTypeElements(elems);}
        ;

recordField returns [ RecordField result = new RecordField() ]
        : i=identifier ':'                      {result.setTheID($i.text);}
          t=componentType                       {result.setTheType($t.result);}
                                        {result.setTheSelection(
                                           new RegionSelection(
                                             $i.text.getTheSelection().getStart(),
                                             $t.result.getTheSelection().getEnd(),false));}
        ;
```

```
recordType returns [ RecordTypeDeclaration result = new RecordTypeDeclaration() ]
@init{ ArrayList<RecordField> fields = new ArrayList<RecordField>();}
        : 'record'
          r=recordField                 {fields.add($r.result);}
          (',' r=recordField                    {fields.add($r.result);}
          )*
          'end' 'record'                                {result.setTheRecordFields(fields);}
        ;

nameDeclaration returns [ NameDeclaration result = new NameDeclaration() ]
        : i=identifier ':'                      {result.setTheID($i.text);}
          t=componentType                       {result.setTheType($t.result);}
                                        {result.setTheSelection(
                                           new RegionSelection(
                                             $i.text.getTheSelection().getStart(),
                                             $t.result.getTheSelection().getEnd(),false));}
        ;


constantDeclarations returns
                [ ArrayList<ConstantDeclaration> result = new ArrayList<ConstantDeclaration>() ]
        : (c=constantDeclaration                {result.add($c.result);}
          )*
        ;

constantDeclaration returns [ConstantDeclaration result = null]
@init { InternalConstantDeclaration internal = new InternalConstantDeclaration();}
        : l1='constant' i=identifier ':' t=componentType
        ( ':=' l=literal        {internal.setTheLiteralValue($l.result);}
                                {result=internal;}
        | 'is' 'external'       {result= new ExternalConstantDeclaration();}
        )
        ';'                     {result.setTheID($i.text);
                                result.setTheType($t.result);}
        ;

nodeDeclarations returns
                [ ArrayList<NodeDeclaration> result = new ArrayList<NodeDeclaration>() ]
        : (n=nodeDeclaration                    {result.add($n.result);}
          )*
        ;

nodeDeclaration returns [ NodeDeclaration result = null ]
@init{  InternalNodeDeclaration internal = new InternalNodeDeclaration();}
        : l1='node' i=identifier
          in=nodeParameters 'returns' out=nodeParameters
          'is'
          ( 'external'                          {result = new ExternalNodeDeclaration();}
          | (l=localDeclarations        {internal.setTheLocalDeclarations($l.result);}
            )?
            'begin'
            s=statementList             {internal.setTheStatements($s.result);}
                                        {result = internal;}
            'end' 'node'
          )
          l2=';'                                {result.setTheID($i.text);}
                                        {result.setTheInParameters($in.result);}
                                        {result.setTheOutParameters($out.result);}
                                        {result.setTheSelection(
                                        new RegionSelection($l1.line,$l1.pos,$l2.line,$l2.pos,false));}
        ;

nodeParameters returns
                [ArrayList<NameDeclaration> result = new ArrayList<NameDeclaration>() ]
        : '(' ')'
        | '(' n=nameDeclaration                         {result.add($n.result);}
          (',' n=nameDeclaration                        {result.add($n.result);}
          )* ')'
        ;

localDeclarations returns
                [ArrayList<NameDeclaration> result = new ArrayList<NameDeclaration>() ]
        : 'local'
          n=nameDeclaration ';'                 {result.add($n.result);}
          (n=nameDeclaration ';'                        {result.add($n.result);}
          )*
        ;

statementList returns [ArrayList<Statement> result = new ArrayList<Statement>() ]
        : (
            ( a=assignmentStatement                     {result.add($a.result);}
            | m=matchStatement                  {result.add($m.result);}
            | f=ifStatement                     {result.add($f.result);}
            )*
          | s = skipStatement                   {result.add($s.result);}
          )
        ;
```

```
skipStatement returns [ SkipStatement result = new SkipStatement() ]
        : l1='skip' l2=';'
                {result.setTheSelection(
                        new RegionSelection($l1.line,$l1.pos,$l2.line,$l2.pos,true));}
        ;


assignmentStatement returns
                [ AssignmentStatement result = new AssignmentStatement() ]
@init {ArrayList<Name> names = new ArrayList<Name>();}
        :
           (n=nameIdentifier                    {names.add($n.result);}
             (',' n=nameIdentifier                      {names.add($n.result);}
              )*
            )

          ':=' e=expression l=';'                       {result.setTheExp($e.result);}
                                        {result.setTheNames(names);}
                                        {result.setTheSelection(
                                          new RegionSelection(
                                            names.get(0).getTheSelection().getStart(),
                                            new Caret($l.line,$l.pos),false));}
        ;

ifStatement returns [ IfStatement result = new IfStatement() ]
@init {IfStatement current = null;}
        : l1='if' e=expression                  {result.setTheExp($e.result);}
          'then' s=statementList                        {result.setTheStatements($s.result);}
                                        {result.setTheSelection(
                                                new RegionSelection(
                                                        new Caret($l1.line,$l1.pos),
                                                                $s.result.get($s.result.size()-1)
                                                                        .getTheSelection().getEnd(),false));}
                                        {current = result;}
          ( l1='elseif' e=expression
            'then' s=statementList      {ElseIfStatement elseIfStatement = new ElseIfStatement();
                                        IfStatement nestedIfStatement = new IfStatement();
                                        elseIfStatement.setTheIfStatement(nestedIfStatement);
                                         nestedIfStatement.setTheExp($e.result);
                                         nestedIfStatement.setTheStatements($s.result);
                                         nestedIfStatement.setTheSelection(
                                                new RegionSelection(
                                                        new Caret($l1.line,$l1.pos),
                                                                $s.result.get($s.result.size()-1)
                                                                        .getTheSelection().getEnd(),false));
                                         current.setTheOptionalElse(elseIfStatement);
                                         current = nestedIfStatement;}
          )*
          (l1='else' s=statementList    {ElseStatement elseStatement = new ElseStatement();
                                         elseStatement.setTheStatements($s.result);
                                         elseStatement.setTheSelection(
                                                new RegionSelection(
                                                        new Caret($l1.line,$l1.pos),
                                                                $s.result.get($s.result.size()-1)
                                                                        .getTheSelection().getEnd(),false));
                                         current.setTheOptionalElse(elseStatement);}
          )?
          'end' 'if' ';'
        ;

component returns [Component result = null]
        : ( i=identifier                {NamedComponent namedComponent = new NamedComponent();
                                         namedComponent.setTheID($i.text);
                                         result = namedComponent;}
                                        {result.setTheSelection($i.text.getTheSelection());}
          |l1= '_'                              {result = new Wildcard();
                                         result.setTheSelection(
                                                new RegionSelection(
                                                        new Caret($l1.line,$l1.pos),
                                                        new Caret($l1.line, $l1.pos + 1),true));}
          )
        ;

matchOption returns [ MatchOption result = null]
@init{  SingleOption tempSingle = new SingleOption();
        ListOption tempList = new ListOption();
        UnionOption tempUnion = new UnionOption();
        RegionSelection selection = new RegionSelection(false);}
        : ( l1= 'nil'                           {result = new ListOption();
                                         selection = new RegionSelection(
                                                new Caret($l1.line,$l1.pos),
                                                new Caret($l1.line, $l1.pos + 3),true);}
          | c= component                        {tempSingle.setTheComponent($c.result);}
                                        {tempList.setTheFirstOptionalComponent($c.result);}
                                        {result = tempSingle;
                                         selection.setStart($c.result.getTheSelection().getStart());
                                         selection.setEnd($c.result.getTheSelection().getEnd());}
             (  '::'                              {result = tempList;}
               (c= component    {tempList.setTheSecondOptionalComponent($c.result);
                                        selection.setEnd($c.result.getTheSelection().getEnd());}
               )
             )?
          |  '\''u=identifier                     {result = tempUnion;
                                         tempUnion.setTheUnionID($u.text);
                                         selection.setStart($u.text.getTheSelection().getStart());
                                         selection.setEnd($u.text.getTheSelection().getEnd());}
            ( c=component                         {tempUnion.setTheOptionalComponent($c.result);
                                         selection.setEnd($c.result.getTheSelection().getEnd());}

            )?
          )                              {result.setTheSelection(selection);}
        ;

matchStatement returns [ MatchStatement result = new MatchStatement() ]
@init{ArrayList<MatchClause> clauses = new ArrayList<MatchClause>();}
        : l1='match' e=expression 'with'                  {result.setTheExp($e.result);}
          m=matchOption '=>' s=statementList     {MatchClause clause = new MatchClause();
                                         clause.setTheMatchOption($m.result);
                                         clause.setTheStatements($s.result);
                                         clause.setTheSelection(
                                           new RegionSelection(
                                             $m.result.getTheSelection().getStart(),
                                             $s.result.get(
                                                $s.result.size()-1).getTheSelection().getEnd(),false));
                                         clauses.add(clause);}
          ( '|' m=matchOption '=>' s=statementList       {MatchClause clause = new MatchClause();
                                         clause.setTheMatchOption($m.result);
                                         clause.setTheStatements($s.result);
                                         clause.setTheSelection(
                                           new RegionSelection(
                                             $m.result.getTheSelection().getStart(),
                                             $s.result.get($s.result.size()-1).getTheSelection().getEnd(),false));
                                         clauses.add(clause);}
          )*
          'end' 'match' l2=';'                  {result.setTheMatchClauses(clauses);}
                                        {result.setTheSelection(
                                                new RegionSelection($l1.line,$l1.pos,$l2.line,$l2.pos,false));}
        ;

expression returns [ Exp result = null ]
        : w=whenExpression                       {result = $w.result;}
          ;


whenExpression returns [Exp result = null]
@init{  WhenExp exp = new WhenExp();
               RegionSelection selection = new RegionSelection(false);}
        : e= defaultExpression                   {result =$e.result;}
        (
          'when' e2=expression                   {exp.setTheRightExp($e2.result);}
                                        {exp.setTheLeftExp(result);}
                                        {selection.setStart($e.result.getTheSelection().getStart());}
                                        {selection.setEnd($e2.result.getTheSelection().getEnd());
                                         exp.setTheSelection(selection);
                                         result=exp;}
        )?
        ;

defaultExpression returns [Exp result = null]
@init{  DefaultExp exp = new DefaultExp();
               RegionSelection selection = new RegionSelection(false);}
        : e= appendExp                   {result =$e.result;}
        (
          'default' e2=defaultExpression                {exp.setTheRightExp($e2.result);}
                                        {exp.setTheLeftExp(result);}
                                        {selection.setStart($e.result.getTheSelection().getStart());}
                                        {selection.setEnd($e2.result.getTheSelection().getEnd());
                                         exp.setTheSelection(selection);
                                         result=exp;}
        )?
        ;

appendExp returns [Exp result = null]
@init{  AppendList exp = new AppendList();
               RegionSelection selection = new RegionSelection(false);}
        :
         e= consExp                              {result =$e.result;}
        (
          '@'  e2=appendExp                      {exp.setTheRightExp($e2.result);}
                                        {exp.setTheLeftExp(result);}
                                        {selection.setStart($e.result.getTheSelection().getStart());}
                                        {selection.setEnd($e2.result.getTheSelection().getEnd());
                                         exp.setTheSelection(selection);
                                         result=exp;}
        )?
        ;

consExp returns [Exp result = null]
@init{  ConsList exp = new ConsList();
                RegionSelection selection = new RegionSelection(false);}
        :
        e= binaryBooleanExp                     {result =$e.result;}
        (
           '::' e2=consExp                      {exp.setTheRightExp($e2.result);}
                                        {exp.setTheLeftExp(result);}
                                        {selection.setStart($e.result.getTheSelection().getStart());}
                                        {selection.setEnd($e2.result.getTheSelection().getEnd());
                                         exp.setTheSelection(selection);
                                         result=exp;}
        )?
        ;


binaryBooleanExp returns [Exp result = null]
@init{  BinaryExp exp = new BinaryExp();
                RegionSelection selection = new RegionSelection(false);}
        :e= binaryRelationalExp                 {result =$e.result;}
        (
          (   'and'                             {exp.setTheBinaryOp(BinaryOp.AND);}
            | 'or'                              {exp.setTheBinaryOp(BinaryOp.OR);}
          )                             {selection.setStart($e.result.getTheSelection().getStart());}
          e= binaryBooleanExp                   {exp.setTheRightExp($e.result);}
                                        {exp.setTheLeftExp(result);}
                                        {selection.setEnd($e.result.getTheSelection().getEnd());
                                         exp.setTheSelection(selection);
                                         result=exp;}
        )?
        ;

binaryRelationalExp returns [Exp result = null]
@init{  BinaryExp exp = new BinaryExp();
                RegionSelection selection = new RegionSelection(false);}
        :e= binaryNumericExp2                   {result =$e.result;}
        (
          ( '='                         {exp.setTheBinaryOp(BinaryOp.EQUAL);}
          | '/='                                {exp.setTheBinaryOp(BinaryOp.NOT_EQUAL);}
          | '<='                                {exp.setTheBinaryOp(BinaryOp.LESS_EQUAL);}
          | '>='                                {exp.setTheBinaryOp(BinaryOp.GREATER_EQUAL);}
          | '<'                         {exp.setTheBinaryOp(BinaryOp.LESS);}
          | '>'                         {exp.setTheBinaryOp(BinaryOp.GREATER);}
          )                             {selection.setStart($e.result.getTheSelection().getStart());}
          e= binaryRelationalExp                        {exp.setTheRightExp($e.result);}
                                        {exp.setTheLeftExp(result);}
                                        {selection.setEnd($e.result.getTheSelection().getEnd());
                                         exp.setTheSelection(selection);
                                         result=exp;}
        )?
        ;


binaryNumericExp2 returns [Exp result = null]
@init{  BinaryExp exp = new BinaryExp();
                RegionSelection selection = new RegionSelection(false);}
        :e= binaryNumericExp1                   {result =$e.result;}
        (
          ( '+'                         {exp.setTheBinaryOp(BinaryOp.ADD);}
          | '-'                         {exp.setTheBinaryOp(BinaryOp.SUB);}
          )                             {selection.setStart($e.result.getTheSelection().getStart());}
          e= binaryNumericExp2                  {exp.setTheRightExp($e.result);}
                                        {exp.setTheLeftExp(result);}
                                        {selection.setEnd($e.result.getTheSelection().getEnd());
                                         exp.setTheSelection(selection);
                                         result =exp;}
        )?
        ;
        binaryNumericExp1 returns [Exp result = null]
@init{  BinaryExp exp = new BinaryExp();
                RegionSelection selection = new RegionSelection(false);}
        : e= expCombined                        {result= $e.result;}
         (
          ( '/'                         {exp.setTheBinaryOp(BinaryOp.DIV);}
          | '%'                         {exp.setTheBinaryOp(BinaryOp.MOD);}
          | '*'                         {exp.setTheBinaryOp(BinaryOp.MUL);}
          )                             {selection.setStart($e.result.getTheSelection().getStart());}

          e2= binaryNumericExp1                 {exp.setTheRightExp($e2.result);}
                                        {exp.setTheLeftExp(result);}
                                        {selection.setEnd($e2.result.getTheSelection().getEnd());
                                         exp.setTheSelection(selection);
                                         result = exp;}

          )?
        ;
```

```
expCombined returns [Exp result = null]
        :  o=unaryExpression                    {result = $o.result;}
         | e=existsExpression                   {result = $e.result;}
         | n=expName                            {result = $n.result;}
         | c=expConstruction                    {result = $c.result;}
        ;

simpleExpression returns [ Exp result = null ]
        : c=callExpression                      {result = $c.result;}
         | p=parenExpression                    {result = $p.result;}
         | l=literal                            {result = $l.result;}
         //NameIdentifier needs to be here for backtracking to work.
         | n=nameIdentifier                     {NameExp name = new NameExp();
                                         name.setTheName($n.result);
                                         name.setTheSelection($n.result.getTheSelection());
                                         result = name;}
        ;

recordAccess returns [ RecordAccess result = new RecordAccess()]
@init{  RegionSelection selection = new RegionSelection(false);}
        : s=simpleExpression   '.' r=identifier          {result.setTheID($r.text);}
                                        { result.setTheExp($s.result);
                                          selection.setStart($s.result.getTheSelection().getStart());
                                          selection.setEnd($r.text.getTheSelection().getEnd());
                                          result.setTheSelection(selection);}
        ;
tupleAccess returns [ TupleAccess result = new TupleAccess()]
@init{  RegionSelection selection = new RegionSelection(false);}
        : s=simpleExpression '#' t=NUMERIC_LITERAL
                                        {result.setTheInteger(new Integer($t.text));}
                                        { result.setTheExp($s.result);
                                          selection.setStart($s.result.getTheSelection().getStart());
                                          selection.setEnd(new Caret($t.line,$t.pos + $t.text.length()));
                                          result.setTheSelection(selection);}
        ;

expName returns [Exp result = null]
        : r=recordAccess                        {result = $r.result;}
        | t=tupleAccess                 {result = $t.result;}
        | s=simpleExpression                    {result = $s.result;}
        ;

expConstruction returns [Exp result = null]
        :  r=recordConstruction                         {result = $r.result;}
         | lc=listConstruction                  {result = $lc.result;}
         | u=unionConstruction                  {result = $u.result;}
         | t=tupleConstruction                  {result = $t.result;}
         ;







unaryExpression returns [ UnaryExp result = new UnaryExp() ]
@init{  RegionSelection selection = new RegionSelection(false);}
        : ( l='not'                             {result.setTheUnaryOp(UnaryOp.NOT);}
          | l='+'                               {result.setTheUnaryOp(UnaryOp.POS);}
          | l='-'                               {result.setTheUnaryOp(UnaryOp.NEG);}
          )                             {selection.setStart(new Caret($l.line,$l.pos));}
          e=expression                  {result.setTheExp($e.result);}
                                        {selection.setEnd($e.result.getTheSelection().getEnd());
                                         result.setTheSelection(selection);}
        ;
existsExpression returns [ ExistsExp result = new ExistsExp() ]
@init{  RegionSelection selection = new RegionSelection(false);}
        : l='exists' e=expression                       {selection.setStart(new Caret($l.line,$l.pos));}
                                        {result.setTheExp($e.result);}
                                        {selection.setEnd($e.result.getTheSelection().getEnd());
                                         result.setTheSelection(selection);}
        ;

callExpression returns [ CallExp result = new CallExp() ]
@init{  ArrayList<Exp> exps = new ArrayList<Exp>();
        RegionSelection selection = new RegionSelection(false);}
        : p=pathIdentifier                      {result.setTheID($p.result);}
                                        {result.setTheOptionalPackageName($p.parent);}
                                        {selection.setStart($p.result.getTheSelection().getStart());}
        '(' (e=expression                       {exps.add($e.result);}
        (',' e=expression                       {exps.add($e.result);}
        )* )?
        l=')'
                                        {result.setTheParameters(exps);}
                                        {selection.setEnd(new Caret($l.line, $l.pos+1));
```

```
                                         result.setTheSelection(selection);}
        ;

parenExpression returns [ Exp result = null ]
        : '(' e=expression ')'                  {result = $e.result;}
        ;

recordConstruction returns [RecordConstruction result = new RecordConstruction() ]
@init{  ArrayList<RecordAssignment> recs = new ArrayList<RecordAssignment>();
        RegionSelection selection = new RegionSelection(false);}
        : p=pathIdentifier
         '\''
         '['                            {result.setTheID($p.result);}
                                        {result.setTheOptionalPackageName($p.parent);}
                                        {selection.setStart($p.result.getTheSelection().getStart());}
          (i=identifier '=>' e=expression)              {RecordAssignment rec = new RecordAssignment();
                                         rec.setTheID($i.text);
                                         rec.setTheExp($e.result);
                                         recs.add(rec);}
          (',' i=identifier '=>' e=expression   {RecordAssignment rec = new RecordAssignment();
                                         rec.setTheID($i.text);
                                         rec.setTheExp($e.result);
                                         recs.add(rec);}
          )*
          l=']'                         {result.setTheRecordAssignments(recs);}
                                        {selection.setEnd(new Caret($l.line, $l.pos+1));
                                         result.setTheSelection(selection);}
        ;

listConstruction returns [ ListConstruction result = new ListConstruction() ]
@init{  ArrayList<Exp> exps = new ArrayList<Exp>();
        RegionSelection selection = new RegionSelection(false);}
        // listing elements of list
        : p=pathIdentifier                      {result.setTheID($p.result);}
                                        {result.setTheOptionalPackageName($p.parent);}
                                        {selection.setStart($p.result.getTheSelection().getStart());}
         '\''
          ('{' (e=expression                    {exps.add($e.result);}
             ( ',' e=expression                 {exps.add($e.result);}
             )*)?
           l='}'                        {selection.setEnd(new Caret($l.line, $l.pos+1));}

           | n='nil'                    {selection.setEnd(new Caret($n.line, $n.pos+3));}
          )

                                        {result.setTheExps(exps);
                                         result.setTheSelection(selection);}
         ;

unionConstruction returns [UnionConstruction result = new UnionConstruction() ]
@init{  RegionSelection selection = new RegionSelection(false);}
        : p=pathIdentifier                      {result.setTheOptionalPackageName($p.parent);}
                                        {result.setTheID($p.result);}
                                        {selection.setStart($p.result.getTheSelection().getStart());}
         '\''
         i=identifier                   {result.setTheUnionID($i.text);}
                                        {selection.setEnd($i.text.getTheSelection().getEnd());}
        ( '('
           e=expression                 {result.setTheOptionalExp($e.result);}
           l=')'                                {selection.setEnd(new Caret($l.line, $l.pos+1));}

         )?                             {result.setTheSelection(selection);}
        ;

tupleConstruction returns [TupleConstruction result = new TupleConstruction() ]
@init{  RegionSelection selection = new RegionSelection(false);}
        : (p=pathIdentifier                     {result.setTheOptionalPackageName($p.parent);}
                                        {result.setTheID($p.result);}
                                        {selection.setStart($p.result.getTheSelection().getStart());}
         '\'')

         '('
           e=expression                 {result.getTheExps().add($e.result);}
           (
           ',' e=expression                     {result.getTheExps().add($e.result);}
           )+
         l=')'                          {selection.setEnd(new Caret($l.line, $l.pos+1));}

                                        {result.setTheSelection(selection);}
        ;

literal returns [Literal result = null]
        : s=STRING_LITERAL                      {StringLiteral sl = new StringLiteral();
                                         sl.setTheString($s.text.substring(1,$s.text.length()-1));
                                         sl.setTheSelection(new RegionSelection(
                                                        $s.line,$s.pos,$s.line,$s.pos+$s.text.length(),true));
                                         result = sl;}
        | n=NUMERIC_LITERAL                     {NumericLiteral nl = new NumericLiteral();
```

```
                                          nl.setTheNumberString($n.text);
                                          nl.setTheSelection(new RegionSelection(
                                                         $n.line,$n.pos,$n.line,$n.pos+$n.text.length(),true));
                                          result = nl;}
        | ('\'' c=. '\'')                                {CharacterLiteral cl = new CharacterLiteral();
                                          cl.setTheCharacterString($c.text);
                                          cl.setTheSelection(new RegionSelection(
                                                 $c.line,$c.pos-1,$c.line,$c.pos+2,true));
                                          result = cl;}
        | b='true'                               {result = new TrueBooleanLiteral();
                                          result.setTheSelection(new RegionSelection(
                                                 $b.line,$b.pos,$b.line,$b.pos+$b.text.length(),true));}
        | b='false'                              {result = new FalseBooleanLiteral();
                                         result.setTheSelection(new RegionSelection(
                                                 $b.line,$b.pos,$b.line,$b.pos+$b.text.length(),true));}
        ;

nameIdentifier returns [ Name result = null ]
        : i=identifier                           { IDName n = new IDName();
                                           n.setTheID($i.text);
                                           n.setTheSelection($i.text.getTheSelection());
                                           result = n;}
        ;


packageIdentifier returns [ PackageName result = new PackageName() ]
@init { ArrayList<ID> path = new ArrayList<ID>();
        RegionSelection selection = new RegionSelection(true);}
        : i=identifier                           {result.setTheID($i.text);}
                                         {selection.setStart($i.text.getTheSelection().getStart());}
                                         {selection.setEnd($i.text.getTheSelection().getEnd());}
          ('.' i=identifier                      {path.add(result.getTheID());}
                                         {result.setTheID($i.text);}
                                         {selection.setEnd($i.text.getTheSelection().getEnd());}
          )*                                     {result.setThePackagePath(path);}
                                         {result.setTheSelection(selection);}
        ;

pathIdentifier returns
        [ OptionalPackageName parent = new EmptyPackageName(), ID result = new ID() ]
@init { ArrayList<ID> path = new ArrayList<ID>();
        RegionSelection selection = new RegionSelection(true);}
        : i=identifier                           {$pathIdentifier.result = $i.text;}
                                         {selection.setStart($i.text.getTheSelection().getStart());}
          ('.' i=identifier                      {if($pathIdentifier.parent instanceof PackageName)
                                               {
                                                 path.add(((PackageName)$pathIdentifier.parent).getTheID());
                                               }
                                          $pathIdentifier.parent = new PackageName();
                                          ((PackageName)$pathIdentifier.parent).setTheID($pathIdentifier.result);}
                                          {selection.setEnd($pathIdentifier.result.getTheSelection().getEnd());}
                                         {$pathIdentifier.result = $i.text;}
          )*                                     {if($pathIdentifier.parent instanceof PackageName)
                                               {
                                                  ((PackageName)$pathIdentifier.parent).setThePackagePath(path);
                                                  ((PackageName)$pathIdentifier.parent).setTheSelection(selection);
                                               }
                                          }

          // parent - identifier
        ;

identifier returns [ID text = new ID() ]
        : i=IDENTIFIER                   {text.setTheIDString($i.text);}
                                         {text.setTheSelection(new RegionSelection(
                                                 $i.line,$i.pos,$i.line,$i.pos+$i.text.length(),true));}
        ;

/*u_identifier returns [UnionID text = new UnionID() ]
        : i=UC_IDENTIFIER                        {text.setTheIDString($i.text);}
                                         {text.setTheSelection(new RegionSelection(
                                                 $i.line,$i.pos,$i.line,$i.pos+$i.text.length(),true));}
        ;
*/

// Guardol 1.0 - Identifiers/fragments
fragment
IDENTIFIER_LETTER
        : UPPER_CASE_IDENTIFIER_LETTER
        | LOWER_CASE_IDENTIFIER_LETTER
        ;

fragment
UPPER_CASE_IDENTIFIER_LETTER
        : 'A' .. 'Z'
        ;

fragment
```

```
LOWER_CASE_IDENTIFIER_LETTER
        : 'a' .. 'z'
        ;

fragment
DIGIT
        : '0' .. '9'
        ;


//
/*LC_IDENTIFIER
        : LOWER_CASE_IDENTIFIER_LETTER
          ( '_'? LETTER_OR_DIGIT )*
        ;

UC_IDENTIFIER
        : UPPER_CASE_IDENTIFIER_LETTER
          ( '_'? LETTER_OR_DIGIT )*
        ;*/

IDENTIFIER
        : IDENTIFIER_LETTER
          ( '_'? LETTER_OR_DIGIT )*
        ;

fragment
LETTER_OR_DIGIT
        : IDENTIFIER_LETTER
        | DIGIT
        ;


//
NUMERIC_LITERAL
        : DECIMAL_LITERAL
        | BASED_LITERAL
        ;


//
fragment
DECIMAL_LITERAL
        : NUMERAL ( ( '.' DIGIT ) => '.' NUMERAL )? EXPONENT?
        ;

fragment
NUMERAL
        : DIGIT ( '_'? DIGIT )*
        ;

fragment
EXPONENT
        : 'E' ( '+' | '-' )? NUMERAL
        ;


//
fragment
BASED_LITERAL
        : BASE '#' BASED_NUMERAL '#' EXPONENT?
        ;

fragment
BASE
        : NUMERAL
        ;

fragment
BASED_NUMERAL
        : EXTENDED_DIGIT ( '_'? EXTENDED_DIGIT )*
        ;

fragment
EXTENDED_DIGIT
        : DIGIT | 'A' .. 'F'
        ;

//This is really annoying - cannot be separated
//CHARACTER_LITERAL_OR_QUOTE
//      : ('\'' . '\'') => '\'' . '\'' | '\''
//      ;

STRING_LITERAL
        : '"' ( '""' | ~('"') )* '"'
        ;
```

```
COMMENT
        :  '//'
           ~( '\n' | '\r' )*
           '\r'? '\n'                            { $channel=HIDDEN; }
        ;

WS
        :   ( ' ' | '\r' | '\t'
            | '\u000C' | '\n' )                  { $channel=HIDDEN; }
        ;
```

Appendix C

Guardol AST Model

```
/******************************************************************************
 * Copyright (c) 2009 Jonathan Hoag, Kansas State University, and others.     *
 * All rights reserved. This program and the accompanying materials           *
 * are made available under the terms of the Eclipse Public License v1.0      *
 * which accompanies this distribution, and is available at                   *
 * http://www.eclipse.org/legal/epl-v10.html                                  *
 *                                                                            *
 * Contributors:                                                              *
 *     Jonathan Hoag - initial API and implementation                         *
 ******************************************************************************/

 /**
  * Guardol Model
  *
  * @author Jonathan Hoag
  */

@Profile org::sireum::profile::modeling::classdesign

@Top            :Node
@Visitor        (packageName = org::sireum::profile::guardol::model,
                          visitorName = :NodeVisitor,
                          clonerName = :NodeCloner,
                          comparatorName = :NodeComparator,
                          @AvoidVisitAttribute :theOptionalParent, :theIDStrings,
                            :theSelection, :theAndTypeDeclarations,
                          @AvoidCompareAttribute :theSelection, :theOptionalParent,
                          @AvoidSubstituteAttribute :theSelection)

package org::sireum::profile::guardol::model;

//The generic AST Node
//Contains:
//-A selection object referring to the starting/ending line numbers/char offsets
record abstract Node
{
        org::sireum::profile::guardol::selection::IRegionSelection theSelection
                @Default new org.sireum.profile.guardol.selection.RegionSelection();
}

//The optional ID node serving as a place holder for a ID or EmptyID
record abstract OptionalID extends Node
{

}
```

```

//The ID node
//Contains:
//-A string object
record ID extends OptionalID
{
        String theIDString
                @Default "";
}
//The empty ID node - No ID present
record EmptyID extends OptionalID
{

}

//The base node for a guardol program file.
//Contains:
//-A list of package declarations
record Compilation extends Node
{
        PackageDeclaration[] thePackageDeclarations
                @NonNull;
}

//The Package Declaration
//Contains:
//-A package name (e.g. Foo.Bar.Pack)
//-A list of with Packages
//-A list of Type Declarations
//-A list of Node Declarations
record PackageDeclaration extends Node
{
        PackageName thePackageName;
        PackageName[] theWithPackageNames
                @Default ^[];
        TypeDeclaration[] theTypeDeclarations
                @Default ^[];
        NodeDeclaration[] theNodeDeclarations
                @Default ^[];
}

//The Qualifier enumeration
//Contains:
//-An element in the enumeration signifying partial or total
enum Qualifier
{
        PARTIAL, TOTAL
}

//Name Declaration is used when defining parameters or variables
//Contains:
//-An id string identifying a variable/parameter name
//-A type identifying the type of the variable/parameter
record NameDeclaration extends Node
{
        ID theID;
        Type theType;
}

//Type Declarations
//The Type Declaration place holder
record abstract TypeDeclaration extends Node
{
        ID[] theAndTypeIDs
                @Default ^[];
        ID theID;
}

record abstract OptionalType extends Node{

}

record EmptyType extends OptionalType{

}

record abstract Type extends OptionalType{
        Qualifier theQualifier
                @Default Qualifier.PARTIAL;
}

record SimpleType extends Type{
        ID theTypeID;
        OptionalPackageName theOptionalPackageName
                @Default new EmptyPackageName();
}

record TupleType extends Type{
        SimpleType[] theSimpleTypes
                @Default ^[];
}

//Abstract Type Declaration
record AbstractTypeDeclaration extends TypeDeclaration
{

}

//Simple Type Declaration
//Contains:
//-A simple type identifying the component type
record SimpleTypeDeclaration extends TypeDeclaration
{
        SimpleType theSimpleType;
}

record TupleTypeDeclaration extends TypeDeclaration
{
        TupleType theTupleType;
}

//List Type Declaration
//Contains:
//-A type identifying the component type
record ListTypeDeclaration extends TypeDeclaration
{
        Type theType;
}

//Record Field
record RecordField extends Node
{
        ID theID;
        Type theType;
}

//Record Type Declaration
//Contains:
//-A List of Record Fields each identifying a field name and a type
record RecordTypeDeclaration extends TypeDeclaration
```

```
{
        RecordField[] theRecordFields
                @Default ^[];
}

//Union Type Element
//Contains:
//-An id string identifying the name of the datatype
//-An optional type identifying the component type of the datatype
record UnionTypeElement extends Node
{
    ID theID;
        OptionalType theOptionalType
                @Default new EmptyType();
}

//Union Type Declaration
//Contains:
//-A list of Union Type Elements containing each datatype definition
record UnionTypeDeclaration extends TypeDeclaration
{
        UnionTypeElement[] theUnionTypeElements
                @Default ^[];
}

//Node Declaration
//Contains:
//-An id string identifying the name of the node
//-A list of in parameters with their ids and types
//-A list of out parameters with their ids and types
//--***IMPORTANT***
//--External and Internal both inherit this class and its fields
record abstract NodeDeclaration extends Node
{
        ID theID;
        NameDeclaration[] theInParameters
                @Default ^[];
        NameDeclaration[] theOutParameters
                @Default ^[];
}

//External Node Declaration
//-Inherits all needed fields from Node Declaration
record ExternalNodeDeclaration extends NodeDeclaration
{

}

//Internal Node Declaration
//Contains:
//-A list of local variable declarations and their types
//-A list of statements
//--NOTE: the statement list could also be composed of one skip statement
record InternalNodeDeclaration extends NodeDeclaration
{
        NameDeclaration[] theLocalDeclarations
                @Default ^[];
        Statement[] theStatements
                @Default ^[];
}

//Statement
//--Abstract definition of a statement
record abstract Statement extends Node
{
}

//Skip Statement
//--An empty statement that does nothing
//--if there is a skip statement then there are no other
//--statements in the statement list (*enforced by ANTLR grammar*)
//--useful for match statements
record SkipStatement extends Statement
{

}

//Assignment Statement
//Contains:
//-A list of names identifying the assignment of the exp to these variables
//-An exp that is used for the assignment
record AssignmentStatement extends Statement
{
        Name[] theNames
                @Default ^[];
        Exp theExp;
}

//If Statement

record abstract OptionalIfStatement extends Statement
{

}

record IfStatement extends OptionalIfStatement
{
        OptionalExp theOptionalExp
                @Default new EmptyExp();
        Statement[] theStatements
                @Default ^[];
        OptionalIfStatement theOptionalElse
                @Default new EmptyIfStatement();
}

record EmptyIfStatement extends OptionalIfStatement
{

}

//Match Statement

//A Component is a wildcard or newly named variable
record abstract Component extends OptionalComponent
{

}

record abstract OptionalComponent extends Node
{

}

record EmptyComponent extends OptionalComponent
{

}

record Wildcard extends Component
```

```
{

}

record NamedComponent extends Component
{
        ID theID;

}

record ListOption extends MatchOption
{
        OptionalComponent theFirstOptionalComponent
                @Default new EmptyComponent();
        OptionalComponent theSecondOptionalComponent
                @Default new EmptyComponent();
}

record UnionOption extends MatchOption{
    Component theFirstComponent;
    Component theSecondComponent;
}

record SingleOption extends MatchOption
{
        Component theComponent;
}

//Match Option
//--A match option is a single case in the matching
//--There is an optional Union type for datatype matching

record abstract MatchOption extends Node
{

}

//Match Clause
//--combines the case with the succeeding statements
//--the statement list can be made of a single skip statement if
//--nothing is to be done in that case
record MatchClause extends Node
{
        MatchOption theMatchOption;
        Statement[] theStatements
                @Default ^[];
}

//Match Statement
//Contains:
//-the exp is element that is being matched against
//-the match clauses are all the cases and their succeeding statements
record MatchStatement extends Statement
{
        Exp theExp;
        MatchClause[] theMatchClauses
                @Default ^[];

}

//Expressions
//--All expressions will have a type so that we can do type checking
//--this type will be identified after parsing when the symbol table
//--is generated

```

```
record abstract OptionalExp extends Node
{

}

record abstract Exp extends OptionalExp
{

}

record EmptyExp extends OptionalExp
{

}

//Record Name is used only to construct record accesses

record RecordAccess extends Exp
{
        Exp theExp;
        ID theID;
}

record TupleAccess extends Exp
{
        Exp theExp;
        Integer theInteger;
}

record abstract InfixExp extends Exp
{
        Exp theLeftExp;
        Exp theRightExp;
}

record NameExp extends Exp
{
        Name theName;
}

record CallExp extends Exp
{
        OptionalPackageName theOptionalPackageName
                @Default new EmptyPackageName();
        ID theID;
        Exp[] theParameters
                @Default ^[];
}

enum UnaryOp
{
        POS, NEG, NOT
}

record UnaryExp extends Exp
{
        UnaryOp theUnaryOp;
        Exp theExp;
}

record ExistsExp extends Exp
{
        Exp theExp;
}
```

```
enum BinaryOp
{
        AND, OR, XOR, EQUAL, NOT_EQUAL, LESS, LESS_EQUAL, GREATER,
        GREATER_EQUAL, ADD, SUB, MUL, DIV, MOD
}

record BinaryExp extends InfixExp
{
        BinaryOp theBinaryOp;
}

record WhenExp extends InfixExp
{

}

record DefaultExp extends InfixExp
{

}

//Literals
//--All literals are expressions
record abstract Literal extends Exp
{

}

record StringLiteral extends Literal
{
        String theString;
}

record NumericLiteral extends Literal
{
        String theNumberString;
}

record CharacterLiteral extends Literal
{
        String theCharacterString;
}

//Union Construction
//-The optional parent id string identifies the package the union type is in
//-The id string is the name of the type
//-The type id string is the name of the datatype
//-The exp is used to construct the type

record UnionConstruction extends Exp
{
        OptionalPackageName theOptionalPackageName
                @Default new EmptyPackageName();
        ID theID;
        ID theTypeID;
        OptionalExp theOptionalExp
                @Default new EmptyExp();

}


// Record Construction
// The record construction creates a record assigning an expression to each field
record RecordConstruction extends Exp
{
        OptionalPackageName theOptionalPackageName
                @Default new EmptyPackageName();
        ID theID;
        RecordAssignment[] theRecordAssignments
                @Default ^[];

}

// Record assignment keeps track of which field gets assigned to an expression
record RecordAssignment extends Node
{
        ID theID;
        Exp theExp;

}

// List Construction constructs a list
// Note: the expressions must all be of the same type
record ListConstruction extends Exp
{
        OptionalPackageName theOptionalPackageName
                @Default new EmptyPackageName();
        ID theID;
        Exp[] theExps
                @Default ^[];
}

record TupleConstruction extends Exp
{
        OptionalPackageName theOptionalPackageName
                @Default new EmptyPackageName();
        ID theID;
        Exp[] theExps
                @Default ^[];
}

// Append list concatenates two lists together
// Note: the expressions must all be of the same type
record AppendList extends InfixExp
{

}

// Cons list adds the element to the beginning of a list
// Note: the first expression must be of the same type of the second expressions elements
record ConsList extends InfixExp
{

}

// Names

record abstract Name extends Node
{

}

//IDName is used to reference a local variable or parameter

record IDName extends Name
{
        ID theID;

}
```

```
//Package name is used when defining a package name and in with clauses

record abstract OptionalPackageName extends Name
{

}

record EmptyPackageName extends OptionalPackageName
{

}

record PackageName extends OptionalPackageName
{
        ID[] thePackagePath
                @Default ^[];
        ID theID;

}
```