A DEVELOPMENT ENVIRONMENT AND STATIC ANALYSES FOR GUARDOL - A LANGUAGE FOR THE SPECIFICATION OF HIGH ASSURANCE GUARDS

by

JOSIAH DODDS

B.S., Kansas State University, 2008

A THESIS

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
John Hatcliff

Copyright

Abstract

There are a number of network situations where different networks have different security policies and still need to share information. While it is important to allow some data to flow between the two networks, it is just as important that they don't share any data that violates the respective security policies of the networks. Constraints on data sharing are often phrased in terms of classification levels of data (e.g. top secret, secret, public). They might also be stated in terms of the contents of the data (e.g. are there military base names, is the location correct). The software and hardware that works to solve these problems is called Cross Domain Solutions (CDS).

There are a variety of hardware platforms capable of implementing CDS. These platforms are all configured in different ways and they are often proprietary. Not only are there a number of platforms on the market, many are difficult to understand, verify, or even specify.

The Guardol project provides an open, non-proprietary, and domain-specific language for specifying CDS security policies and implementing CDS. Guardol is designed to be easy to understand and verify.

This thesis describes the design and implementation of primary Guardol components. It includes a description of the Eclipse GUI plug-ins that have been developed for the project as well as a description of new formal analyses and translations that have been developed for the language. The translation is used to plug into external tools for model checking and the analyses help to make the translation clean and efficient. The analyses are also useful tools to help make the use of Guardol easier for developers.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Cross Domain Solutions

There are a number of network situations where different networks have different security policies and still need to share information. While it is important to allow some data to flow between the two networks, it is just as important that they don't send any data that the security policy doesn't allow them to send. The data is generally restricted by classification level, but it can also be separated by other factors such as the date it was published, or words that it contains. There are a number of situations in both the military and civilian worlds that require cross domain solutions.

"Cross Domain Solutions (CDS) are controlled interfaces that provide the capability to access or transfer information across different security domains." [1]

One of the most common examples is the military. In the military information has a classification level, and everyone has clearance to view up to a certain level. A domain that is at the top secret level still needs to be able to send messages to a domain at the classified level. The problem is that it should only be able to send messages that are at the classified level or lower. It might also want to decrease the classification level of information by removing certain key words, or even images.

A similar situation is military cooperation between different countries, where each military has it's own network. It is important to control the flow of information between countries

There are also more dynamic situations that occur. One is that soldiers in the field could require intelligence for the area that they are in, but not be cleared to receive information about other areas. In this case, their GPS location could be checked for proximity to the location the intelligence pertains to before the information is released.

The applications are not limited to the military either. One possible civilian use for cross domain solutions is the medical field. A current problem is how to allow various medical institutions to share a patient's medical data in a way that preserves the privacy of the patient. A cross domain solution could be used, and the patient could be allowed to set the policies regarding what information can be released to who.

1.2 Network Guards

Cross domain solutions require something to control the flow of information between domains, and they need to be somewhat intelligent to deal with the possible demands. Guards are hardware that controls the information flow between domains. Each guard generally sits between two different domains and is responsible for carrying out the policies set by their owners.

Most guards have a programming language associated with them. The programming languages for guards can vary from XML to proprietary languages. Most of the languages are built to allow the specific functionality that a guard needs. It is also important that they be able to make calls to external software like a virus checker.

There are a number of currently existing guard platforms such as DFCF (2.2) and Turnstile (2.3). Both use their own programming language

1.3 Current Guards

Almost every current solution for programming network guards is proprietary, this means information can be difficult or impossible to find. It also means that it is difficult for anyone outside of the company that owns the guard to write and maintain guard policies. This means that people that buy guards are forced to pay and wait for the owner of the programming language to write or update policies. This leads to a very slow and expensive process every time a guard needs to be created or changed.

In high security environments like health-care and the military, it is vital that guards do what they are expected to do. It is unacceptable for a guard to only block 99% of top secret information from flowing to a classified domain. It is important to be able to verify that a policy actually does what it is meant to do. Current guard programming languages do not include tools that allow you to verify the code that you write. This means it is possible for security breaking bugs to slip by. Verification of guards usually requires close hand inspections, which are still prone to error.

1.4 Domain Specific Languages

Most of today's popular programming languages are build to be able to solve any type of problem that could be given to them, and be usable in the largest possible number of situations. While this is useful in many cases, often it is more convenient to have a programming language that is specifically designed to solve a certain class of problem. These more specific programming languages are called domain specific languages (DSLs).

There are a number of advantages to using DSLs. Almost all of the advantages stem from a smaller number of commands that can perform more specific tasks. On the programmer's end, it can be much easier to learn to use a DSL because the grammar is much simpler. The language can also perform better, because it can be more heavily optimized. Finally, it can be much easier to prove properties about languages that are more restricted.

1.5 Guardol

Our goal is to implement a DSL for describing guard properties and their implementations. We are calling this language Guardol. Because Guardol is a DSL, it will come with all of the advantages listed in 1.4. However it is important to remember that the scope must remain limited to specifying guard behavior. Guardol will be less specific than some of languages that are currently used to program guards because it will not be tied to any hardware. Instead it will be possible to translate Guardol into a number of languages that are used to program guards.

Because Guardol translations could run on multiple applications and in multiple contexts it is important to make sure that it is able to use the local resources of different hardware appropriately. The language needs to be able to describe operations and types that it doesn't include.

Guardol will also often be deployed in high security environments. In these situations it is important to be sure that Guardol is doing exactly what it says it is doing, and to confirm that the program matches its specification. The language will include tools to check both of these properties. There will be a syntax check to help the programmer write correct programs. There will also be a number of static analysis to check type correctness and other properties the programmer might specify.

1.6 Thesis

The goals of this thesis are to

- Introduce the idea of cross domain solutions and guards
- Show how Guardol is needed in this domain
- Present the Guardol language through an example
- Lay out the current architecture of the Guardol tool-chain.
- Give a formal representation of Guardol's syntax
- Present the Static Analyses performed by the Guardol tools
- Prove the correctness of those Analyses
- Formally describe the translation from Guardol to the Simp target language

This thesis is organized to complete these goals in the order listed.

1.7 Contributions

This thesis presents a number of contributions:

- The design of a DSL in a relatively unexplored domain,
- a compiler with the ability to declare external types values and nodes,
- a translation to a language with different constructs than guardol, and
- a partial total inference system.

The greatest overall contribution that this thesis presents is the design of a DSL in a relatively unexplored domain. As described in Section 1.4 the domain of programming guards is one that will benefit greatly from having it's own DSL. Guardol will allow users of cross domain guards to program and specify guards more quickly, and they will be able to program the guards internally instead of sending them off to another company. The various analyses that will come packaged with Guardol will also give guard users a new level of trust that their guard is actually doing what they want it to.

One aspect that allows Guardol to work in the guard domain is the ability to declare external types, values, and nodes while easily integrating them into the rest of the code base. The Guardol grammar provides the external keyword to allow this. The Guardol type system, static analysis, and translation all must be flexible to allow the use of nodes types and values that might not be completely visible in the code.

This thesis also describes the translation from Guardol to Simp. This is an interesting translation because Guardol allows for undefined values but Simp doesn't. This means that the translation for Simp must simulate undefined values. The only way to do this however, requires an increase in both code size and program complexity. There is no need for the simulation of undefined values if we can show that a variable will never be undefined. We can do this in two ways. We can either allow the programmer to specify what variables should never be undefined and check to see if they are correct, or we can infer which variables will never be undefined. Guardol comes with the ability to do both of these. The first allows programmers to specify their program, and the second allows for improved performance in the translation.

Many of the contributions originated from Dr. John Hatcliff, Dr. Simon Ou, Dr. Torben Amtoft, Rockwell Collins and Jon Hoag. Dr. Hatcliff, Dr. Ou, Rockwell Collins and Jon Hoag contributed to the discussion that led to the design of the formalization of both the language and the analyses. Once that discussion was done Jon Hoag finished the initial implementation of the programming language. Along with Dr. Amtoft, Jon then provided the initial formalization. Everyone mentioned above also contributed to the design of the new formalizations and implementations that are discussed in this thesis.

Chapter 2

Background

2.1 Cross Domain solutions

There are a number of current Cross Domain Solutions. This section will briefly outline a number of them.

- Radiant Mercury (RM) is a network guard developed by Lockheed-Martin. It has been in use as a network guard between different classifications of simulations since 2000. It has participated in two different simulation architectures each with their own requirements. One monitored and sanitized two way data-flow and the other was "guard-only". "Guard-only" means that no data was modified, only passed or not passed. Like turnstile (see section 2.2) the guard operates on name/value data pairs parsed from messages. RM must be configured by trained RM operators but it is able to be quickly configured to an accreditable state for different requirements.[2]

- Tactical Cross Domain Solutions (TCDS) is a CDS by General Dynamics. It utilizes SELinux as a platform, which means it can run on off-the-shelf computer hardware. Its internal structure is modular so specific guard behaviors can be plugged in when they are needed. TCDS is capable of parsing XML with schemas or a customizable engine based parsing for non-XML data-types.[3]

- Owl Enterprise Cross Domain Solution (ECDS) also runs on SELinux. It requires two custom network cards, one send only and one receive only. ECDS can perform virus checks, file format checks, and content examinations. The separate send and receive network cards ensure that no information will be transmitted without permission from the ECDS system. The two cards also form a data diode which means that each ECDS system is one way and guarantees that information can only flow in one direction.[4]

- Tenix's CDS also runs of off the shelf technology. It consists of an Email Transfer Application (ETA) and a Data Forwarding Application (DFA). Each act as data diodes and allow information to flow from low security to high security without fear of confidential information leaking back.[5]

```
102.321.482.219    AND(AND(FIELDS ("Classification" "ReleaseableTo"),
                           ONLY ("Classification", "ReleaseableTo", "Payload"))
                       OR(NOT(EQUAL(Classification,"TOPSECRET")),
                          CONTAINS(ReleaseableTo,"AUS"))
102.321.482.211    AND(AND(FIELDS("Classification","ReleasableTo"),
                           ONLY("Classification","ReleaseableTo","Payload")),
                       EQUAL("Classification","Unclassified"))
```

Figure 2.1: Turnstile Example

- Boeing's Secure Network Service allows for safe two way communication between network domains with different security policies.[6]

If the descriptions of these technologies seem brief, it is because there is very little public information about them. Because information about these products is so well controlled it is often impossible for anyone purchasing them to configure the behavior themselves, or even verify that the guard is actually doing what they want.

2.2 Turnstile

Turnstile is a programmable network guard. Turnstile is hardware that sits between two network domains. It can be either unidirectional or bidirectional. We will discuss unidirectional here, because it can easily be expanded for bidirectional. Unidirectional Turnstile has a message input port, a message output port, and an audit output port.

The programming language that Turnstile guards are programmed in is extremely simple. It composes a table that maps IP addresses to rules. The IP addresses specify the destination computer, and the rules are constraints on the messages that are allowed to travel to that IP address. Figure 2.1 is an example of a basic turnstile program.

There are a number of other hardware components to Tunrstile. The two that we are most interested in are the guard engine and the offload engine. The Guard engine is the hardware that evaluates messages against rules and decides if the message can pass through the guard. There are two offload engines in each Turnstile. The offload engines have the job of interfacing with the network. One offload engine sits on the input port of Turnstile, and the other on the output. The input offload engine has the job of accepting incoming messages and transforming them to a form that the guard engine can operate on. The output offload engine then translates the message back into a form that can be sent over the network (if the guard engine forwarded it on to the destination).

The guard engine performs its operations on a table that maps field names to values. It is also able to access the destination IP. So if Turnstile received an input message encoded in XML as in Figure 2.2 the input offload engine would parse the file and pass the key value mapping in figure 2.3 to the guard engine. In Figure 2.3 Classification and ReleasableTo are keys that map to values "TOPSECRET" and "AUS" respectively.

```
<?xml version="1.0" encoding="UTF-8"?>
<message from="192.168.1.3" to="192.168.1.4" classification="TOPSECRET">
        <releasableto>
                AUS
        </releasableto>
<body>
        Hey there Austrailia!
</body>
</message>
```

Figure 2.2: XML input to turnstile

```
[Classification = "TOPSECRET",
  ReleaseableTo = "AUS"]
```

Figure 2.3: Turnstile key value mappings for XML input

The set of all keys in a message is referred to as the table of contents. The table of contents may not be consistent between messages. This allows messages to change in their format and relay different information depending on the message type or source.

Because message formats can change, the Turnstile rules include the operations `ONLY` and `FIELDS`. `ONLY` operates on a table of contents and a list of strings. It returns `TRUE` if every value of the table of contents is contained in the provided string list.

The `FIELDS` operator also takes a table of contents and a string list as an input. It returns `TRUE` if all of the values in the string list are in the table of contents. These two operators can be seen as upper and lower bounds on a table of contents where `FIELDS` $\subseteq$ table of contents $\subseteq$ ONLY. The example in Figure 2.4 states that the toc for the current message must contain keys "Classification" and "ReleasableTo" and can optionally contain "Payload". The message cannot have any other table of contents keys because only acts as an upper bound.

The only other operation from Figure 2.1 that needs explanation is the `EQUAL` operation. It is important to note that the first string input is the name of a field in the table of contents, and the second string is the value to be compared against. The `EQUAL` operation retrieves the value associated with the key it is given and compares it to the second argument.

```
        AND(FIELDS ("Classification" "ReleaseableTo"),
            ONLY ("Classification", "ReleaseableTo", "Payload"))
```

Figure 2.4: Turnstile code example

2.3 DFCF

DFCF uses XML to describe both domains and guards. The goal of DFCF is to provide a guard independent method for describing cross domain policies.

2.3.1 DFCF File Structure

DFCF stands for Data Flow Configuration File. The DFC file is simply a zip file that contains a number of XML files

1. dfc.xml is a required file containing
   (a) data flow meta-data
   (b) guard info
   (c) filter configuration
   (d) classification labels
   (e) domains
   (f) domain pairs
2. comments.xml is an optional file containing plain text comments
3. digitalsignings.xml is an optional file that contains digital signatures
4. changelog.xml is a required file that keeps track of changes to the other files
5. bray.xml is a required file

These files should be sufficient to build and maintain descriptions of cross domain policies.

2.3.2 Domains

One of the most important concepts in DFCF is domains. A domain is simply a collection of some number of endpoints. An endpoint is a specific machine on a network. Endpoints are generally distinguished by distinct IP addresses. Like the description of domains in 1.1 a domain in DFCF describes a number of endpoints that have their own security policy. Domains are described in the dfc.xml file of DFCF.

The dfc.xml file also contains the descriptions of Domain Pairs. Domain pairs connect two domains that are allowed to send messages between them. Domain pairs don't have direction, they only provide a grouping between domains

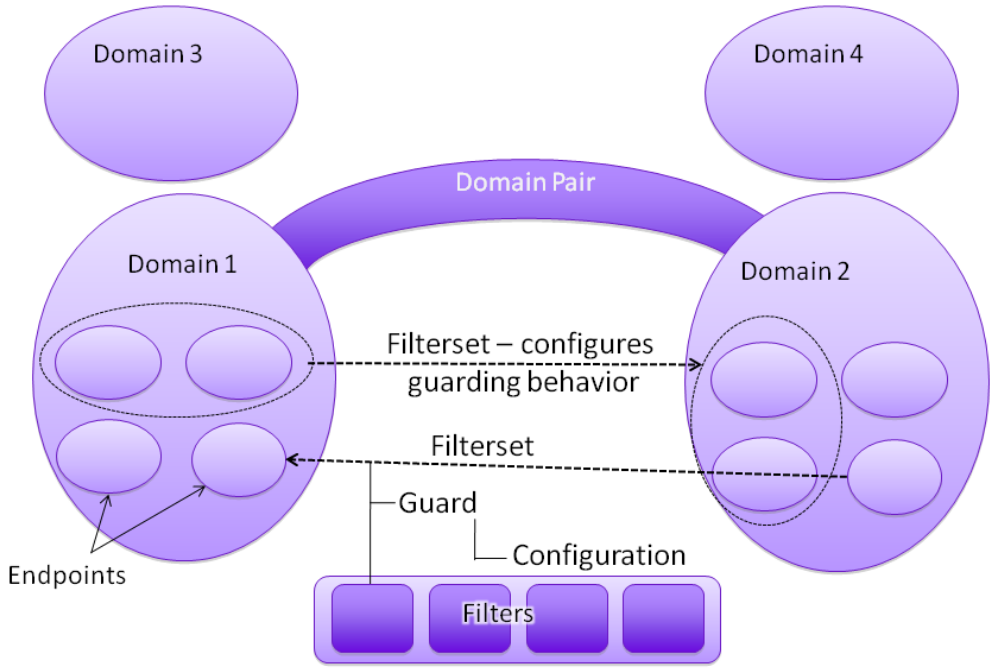

Figure 2.5: Descriptions of DFCF domains and filters

2.3.3 Filterset

Filters are where the policies are actually declared. Each filter describes a single type of functionality. Filters are grouped into filtersets which specify one way communication between two domains that belong to the same domain pair. A filterset can operate on any number of endpoints on each side as long as all of the endpoints on either side belong to the same domain. This can be seen in Figure **??**.

The filterset specifies the guard that enforces it. This gives the unique ability to use different guards for different types of messages, or for messages from different sources. DFCF also allows for the use of a generic guard when it isn't important to use a specific guard. Along with specifying the type of guard ot be used, the file also allows you to specify options for the configuration of the guard.

The part of the filterset that actually specifies it's behavior is the filters. A filterset can contain any number of filters. The filters come in a number of types

1. XML validator - references a schema for validation
2. The Schematron - allows for more in depth XML validation
3. XSL transformer - modifies style information
4. XSL validator - validates style info

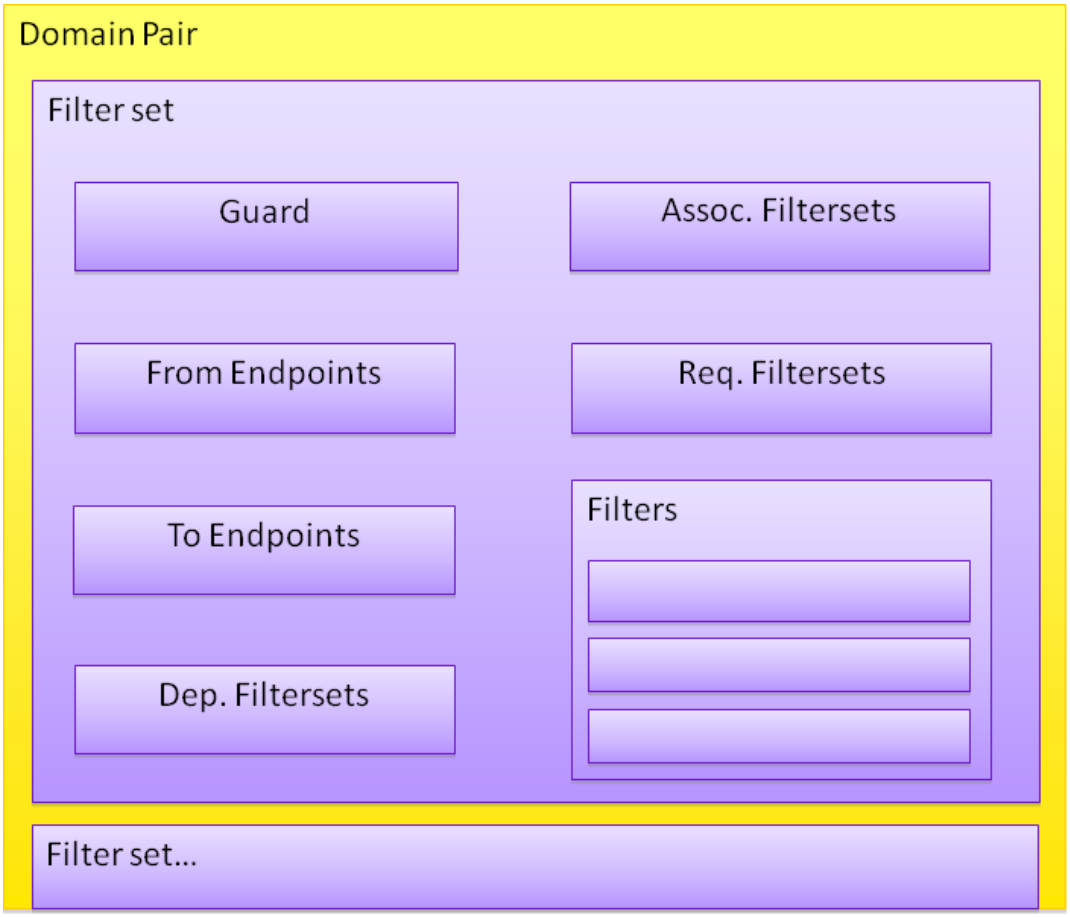

Figure 2.6: Chart of DFCF filtersets

5. word checker - performs a dirty/clean word check
6. XML normalizer - generally used for whitespace cleanup
7. classification checker
8. virus checker
9. digital signature validator
10. digital signature remover
11. custom filters

A single filterset can have any number of any of these filters. When combined they represent all of the behavior that the guard is expected to perform on messages travelling between the two endpoints of the filterset.

Chapter 3

Rationale

3.1 MIME Checker

We will introduce Guardol with an example application. This application checks MIME email and determines if it is safe to cross between domains. The application has the following requirements:

1. A message can consist of a body and some number of attachments
2. A message body is plain text
3. An attachment consists of a name and an object
4. Attachment objects can be other emails, text, XML, binary data, or some other type
5. The body and attachments must be run through a virus check
6. The message body must be checked for a list of "dirty words". These are words that are not allowed to pass from one domain to another
7. Each attachment must be checked
8. Attachments may not have executable extensions
9. If a message is denied or changed for any reason the decision must be reported in an audit message

3.2 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. GIL supports parameterized (generic) types such as `string`. However, the language requires that all types be named.

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

Figure 3.1: Declaration of the *MIME_Type* record

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

Figure 3.2: Declaration of the *AttachmentList* list

We will begin writing the application by defining the data-types that we wish to use. The top level data structure will be the message. As defined above, a message consists of a body and some number of attachments. We will describe this in Guardol with a record (see Figure 8.1). A record is used to group a finite number of data items together. The items can be of mixed types.

Now our message type is set up, but we are missing the type *AttachmentList*. We will use a list to describe our attachments because we don't know how many attachments we might have. A list is the only data-structure in Guardol that doesn't have a set size. The declaration of *AttachmentList* is in Figure 3.2.

Finally we need to set the type of *Attachment*. We know an attachment is simply a name and an object, so we declare another record to hold these two values. The attachment name is a string, but we need to use a union as the type of the object. A union is for when there are multiple options for a value. The values may or may not have types associated with them. If they do have a type, that means they will be associated with a value of that type at run-time. The declaration of *AttachmentType* is Figure 3.3.

3.3 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

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

Figure 3.3: Declaration of the *AttachmentType* union

```
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. The only non-local variables in Guardol are constants. Figure [] defines the top-level function of checking a MIME message.

```
node MIME_Check
 (Input : MIME_Type total) 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;
```

Figure 3.4: Declaration of the *MIME_Check* 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. Statements can be assignments of values to variables, if statements, or match statements. In GIL, a variable can be assigned at most once. A variable that is never assigned has an undefined value. There are also expressions in GIL that can produce an undefined value. For example, 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 that expression is the right-hand-side of an assignment

statement the left-hand-side variable will still be undefined.

First, 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`.

Second, the node performs a dirty-word on the message body *if virus check succeeded*. The "when" expression will evaluate to the left operand if the right operand evaluates to "true". 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".

Third, 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. We also know that `virusOk`, `dirtyAudit`, and `attachAudit` might contain reasons that the message should be blocked. This information is sufficient to create the outgoing message and the audit message. To construct a value with a type `T`, GIL uses the expression `T'DataContent`. For a record type the corresponding fields are given values in `DataContent`. For union types the chosen branch and associated data value are given in `DataContent`.

The newly constructed MIME message will have `newBody` as its body and `newAttachments` as its attachments. GIL supports a "strict" evaluation semantics. This means that if a component of a construction expression is undefined, the whole expression becomes undefined. Under the strict semantics, the guard will discard the whole message when any component of its components violates the policy. The guard behavior 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 is ok 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 like virus checking and data scrubbing. In these cases the node declaration explicitly uses the `external` keyword and no definition of the body needs to be given. Figure 3.5 is an example of two external node declarations. Figures 3.6, 3.7, and 3.8 contain the complete mime example.

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

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

Figure 3.5: Declaration of two external nodes

```
package MIME is

type XML_DOM_Type is external;

type byte is external;

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 total)
is external;

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

node EXEC_CHECK
```

Figure 3.6: Complete mime example part 1

```
  (input : ByteList) returns
  (ok : bool total)
is external;

node NOT_SUFFIX
  (input : string, suffixes: stringList) returns
  (ok : bool total)
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 total) 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 total) returns
  (audit : AuditMsg, newList : AttachmentList)
is
local
  newHd : Attachment;
  newTl : AttachmentList;
  auditHd : AuditMsg;
  auditTl : AuditMsg;
begin
  match Input with
    hd::tl =>
      auditHd, newHd := attachmentCheck(hd);
      auditTl, newTl := attachmentListCheck(tl);
      newList := (newHd::newTl);
```

Figure 3.7: Complete mime example part 2

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

node attachmentCheck
  (Input : Attachment total) returns
  (Audit : AuditMsg, newAttachment : Attachment)
is
local
  isNotExeSuffix : bool;
  newObject : AttachmentType;
  objectAudit : AuditMsg;
  newMail : MIME_Type;
  newText : string;
  newElem : XML_DOM_Type;
  wellFormed : bool;

begin
  isNotExeSuffix := NOT_SUFFIX(Input.name, stringList'{".exe", ".com", ".dll", ".o", ".vba
  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 objectAudi
  newAttachment := Attachment'[name => Input.name, object => newObject];
end node;

end package;
```

Figure 3.8: Complete mime example part 3

Chapter 4

Tool Architecture

Figure 4.1 lays out the architecture of the Guardol compiler. As you can see the architecture mosly follows a linear path through the tools. We begin with user-created GIL (Guardol Implementation Language). The code is transformed in an automatically generated Antlr parser into a Java AST. This AST is translated into Pilar, which is our intermediate representation of the guardol language. From there we run a number of Static Analyses on the IR. These Analyses don't give any input but are required to generate the SIMP. Finally the pilar can be translated into Simp. Simp is a language created by Rockwell Collins to present an easy way to describe simple programs. Rockwell Collins has the ability to translate Simp into Ada which they can then model check.

This section will provide a brief introduction to each segment of the tool-chain, and chapters 7 and 8 will give a deeper look at two parts of the compiler.

4.1 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. LL* grammars allow grammar productions to be written naturally. Many parser generators don't process this type of grammar. ANTLR is also known for its ability to recover from problems in input and still continue parsing the file. ANTLR also presents the user with error messages that are helpful in fixing problems with the input file. Below is an example production rule written in ANTLR's grammar language.

```
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 ';'{result.setTheExp($e.result);}
  {result.setTheNames(names);}
```

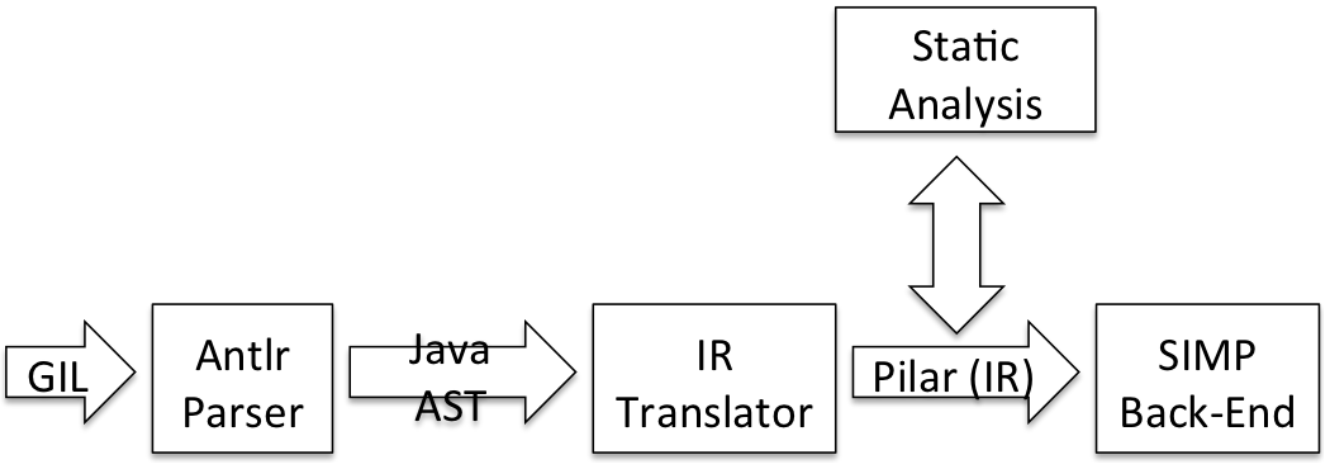

Figure 4.1: Guardol Compiler/Tool-chain Architecture

;

The ANTLR production rule above specifies the rule for parsing and generating the Java Abstract Syntax Tree (AST) for an assignment statement. The Backus-Naur Form (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` are other production rules within the grammar. This BNF rule requires the left hand side of an assignment statement to have one or more (comma separated) `nameIdentifier`. It also declares the assignment operator to be `:=` and requires the right hand side of the statement to be an `expression`.

The code wrapped in curly braces and brackets is used to help ANTLR automatically generate an 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.

ANTLR has been widely used in both academia and industry (some examples are 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.

4.2 Abstract Syntax Tree

As shown in section 4.1, the Antlr grammar contains Java code to generate an AST. An AST is a code representation of a program's syntax. It is useful to us because it puts the

```
//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 4.2: Class Design for `AssignmentStatement` class

program into a normalized form that is easy for us to traverse without having to worry about things like the formatting of the code. The Java classes that make up the Guardol ASTs were generated by a Sireum class design module.

Figure 4.2 gives an example of three Pilar class design records. Each record in a Pilar class design file is responsible for generating a single class in the AST model. The first definition in the figure is the definition of `Node`. This record is used as a top record. This means that every other record in the model will extend the `Node` record. The extension can be either direct, like `Statement` or indirect like `AssignmentStatement`.

If we examine the definition of the `Node` record, we can learn about how Pilar class design works. `Node` is declared abstract, which means the same thing as it does in Java; `Node` can have fields but it may not be instantiated. The record also has a single field. Because the `IRegionSelection` class is not a part of this model it must be referred to by its fully qualified name. The `@Default` clause below is an optional clause declaring what the `theSelection` field should be initialized to. By giving our top node a field we are stating that we want every single record in our model to also have this field. In this case the top record's field is a region selection. This is important because we want to keep the selection information for every piece of information we extract from our input code.

```
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));
```

Figure 4.3: A fragment of Guardol source code

The declaration of `Statement` is not particularly interesting because it doesn't have any fields. The declaration of `AssignmentStatement` however, is not abstract and contains two fields. The first field is declared `Name[] theNames`. `[]` means that the field should be a list of `Names`. `AssignmentStatement` will also have a `theSelection` field that is inherited from `Node`.

The Sireum framework takes a file consisting of record declarations and creates a number of Java class files where each class implements a single record. Sireum also generates a class for traversing the AST. This class implements the well known visitor design pattern. The methods in the generated visitor can be overridden to make it easy to traverse and translate the AST. In our case we override the methods with methods that create a translation to our Intermediate Representation (IR) language.

4.3 Intermediate Representation

The IR Translation phase translates a Guardol AST to an intermediate representation in 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. In three address code form, each instruction implements exactly one operation. For example, a complex statement such as:

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

is translated as two statements:

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

Figure 4.4 presents the Guardol Pilar IR of the example in Figure 4.3.

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.

```
# $temp2 := Input @Loc(startline = 75,startcol = 25,endline = 75,endcol = 30) ;
# $temp1 := MIME::VIRUS_CHECK($temp2)  @SecondaryLoc();
# virusOk @Loc() := $temp1  @SecondaryLoc();
# $temp6 := Input @Loc() ;
# $temp5 := $temp6.body  @SecondaryLoc();
# $temp4 := MIME::DIRTY_WORD_SEARCH($temp5)  @SecondaryLoc();
# $temp7 := virusOk @Loc() ;
# $temp3 := when$$($temp4,$temp7)  @SecondaryLoc();
# newBody @Loc(), dirtyAudit @Loc() := $temp3  @SecondaryLoc();
# $temp10 := Input @Loc() ;
# $temp9 := $temp10.attachments  @SecondaryLoc();
# $temp8 := MIME::attachmentListCheck($temp9)  @SecondaryLoc();
# attachAudit @Loc(), newAttachments @Loc() := $temp8  @SecondaryLoc();
# $temp12 := newBody @Loc() ;
# $temp13 := newAttachments @Loc() ;
# $temp11 := MIME::MIME_Type ^{:body -> $temp12, :attachments -> $temp13}  @SecondaryLoc();
# Output @Loc() := $temp11  @SecondaryLoc();
# $temp15 := attachAudit @Loc() ;
# $temp18 := MIME::AuditMsg$Dirty_Word_Check_Failed()  @SecondaryLoc();
# $temp21 := newBody @Loc() ;
# $temp20 := exists$$($temp21)  @SecondaryLoc();
# $temp19 := !$temp20  @SecondaryLoc();
# $temp17 := when$$($temp18,$temp19)  @SecondaryLoc();
# $temp23 := MIME::AuditMsg$Virus_Check_Failed()  @SecondaryLoc();
# $temp25 := virusOk @Loc() ;
# $temp24 := !$temp25  @SecondaryLoc();
# $temp22 := when$$($temp23,$temp24)  @SecondaryLoc();
# $temp16 := default$$($temp17,$temp22)  @SecondaryLoc();
# $temp14 := default$$($temp15,$temp16)  @SecondaryLoc();
# Audit @Loc() := $temp14  @SecondaryLoc();
# return;
```

Figure 4.4: Pilar intermediate representation (Location information has been removed after the first line to improve readability)

4.4 Type Checker

The Guardol compiler currently implements two different type checkers. The first phase of type checking enforces compliance to the Guardol type system. The type checker is an implementation of a constraint solving algorithm. Type constraints are generated during a traversal of the AST. Once all of the constraints are generated the constraints are solved by a unification algorithm.

The second type checker is responsible for checking that the partial/total constraints of the Guardol language are enforced. This analysis consists of two phases, an assignedness check and a definedness check. The partial/total type checker can also infer when variables can be declared total. The partial/total analysis is discussed in depth in chapter 7.

4.5 Simp Translation

An interesting problem in translating from GIL to Simp is that Simp does not allow variables to be undefined. It is possible to simulate undefined values in Simp but the simulation comes at the cost of efficiency. By definition total variables are always defined when they are used, so they don't require the undefined simulation. This is a major motivation in the partial/total analysis described in section 4.4. The Simp translation becomes easier to read and more efficient as the number of total variables increases. A descriptive example of the Simp translation can be found in chapter 8

Chapter 5

Guardol GUI

To make Guardol easier to use we have built a Graphic User Interface (GUI). We used software called Xtext[7] to create a plug-in for the Eclipse editor. We have a number of goals for the GUI.

- Support editing of Guardol program text with syntax highlighting,
- perform basic syntax analysis and error reporting,
- perform light-weight semantic analysis,
- provide a higher level organizational view of code
- provide other facilities common in modern IDEs such as code completion and declaration linking, and
- serve as a platform for integrating tools that work on Guardol. The compiler is a primary example of this.

We had a choice between building an Eclipse plug-in from scratch or using one of a number of tools that generate plug-ins for DSLs. We went with a tool called Xtext.

5.1 Xtext

Xtext is a language development framework that allows developers to easily create complete Eclipse plug-ins for Domain Specific Languages. It has its own language that defines a grammar for your language. It then uses the grammar to automatically generate an ECore model of your language. ECore is a part of the Eclipse Modeling Framework (EMF) and is compatible with a number of features outside of Xtext. We decided to use Xtext because its grammar input language is similar to the existing grammar we use for our compiler's parser. Just by entering the grammar Xtext immediately gave us a number of the featuers we had hoped to get from a GUI.

The grammar also generates an Antlr parser. The parser reads the program that the user is editing and converts it into an ECore model. The resulting ECore model is an Abstract Syntax Tree (AST). The generated plug-ins then use this model to give the user various graphical and interface assistance. We have already seen that the Guardol compler has a Antlr parser that generates an AST. Unfortunately, the Xtext grammar generates a spearte parser that then generates the Ecore model. The XText generated parser is automatically generated, and very difficult to work with or modify. Because of this issue, we are left with a project that has two different grammars, parsers, and ASTs. This can lead to difficulties when the grammar needs to be changed because it then needs to be changed in two places for an effective update. Any time the grammar needs to change, both the XText grammar file and the Antlr grammar file must be changed, often in slightly different ways. There doesn't seem to be any obvious way to merge the two because the Xtext isn't very flexible in what AST it uses. If we were to swicth the guardol toolchain to the XText AST, the complier would then depend on the eclipse development environment, which is much heavier than we want.

5.1.1 Xtext Grammars

We will discuss how to use Xtext by building part of the Guardol plug-in. Recall that a Guardol type declaration looks like:

```
type my_Int  is int;
```

and a tuple type declaration is:

```
type my_Tuple is int * my_Int;
```

Our goal is to create an Eclipse plug-in that can recognize some number of these type declarations. The Xtext grammar language looks very similar to the way most grammars are defined. Our first rule will allow us to have some number of type declarations. Similar to the way that most grammars would be described, our Xtext looks like:

```
TypeDeclarations :
    (TypeDeclaration)*;
```

"TypeDeclarations :" tells Xtext that we are describing a rule called `TypeDeclarations`. The "*" operator means that there could be any number of `TypeDeclaration`s. This information is sufficient for a parser, but we need to give Xtext a little more information so that it can build a correct model of our language. We can update the example:

```
TypeDeclarations :
        (decs +=TypeDeclaration)*;
```

Now when Xtext looks at our grammar it will know that it needs to create an AST node called TypeDeclarations. It can also tell what fields that Node should have. Because we told it that each TypeDeclaration should be added to decs, Xtext knows that the TypeDeclarations node should have a field of decs and that it should have a collection type. Now Xtext will be giving us an error because we haven't defined the TypeDeclaration rule yet. It will look like this:

```
grammar org.xtext.example.GuardolTypes with org.eclipse.xtext.common.Terminals

generate guardolTypes "http://www.xtext.org/example/GuardolTypes"

TypeDeclarations :
        (decs +=TypeDeclaration)*;

TypeDeclaration :
        'type'
        name=ID
        ( 'is'
                type=Type
        )?
        ';'
;

Type :
        TupleType | SimpleTyle
;

TupleTyle :
        types+=SimpleType
        ('*' types+=SimpleType)*
;

SimpleType :
        ID
        | 'int'
        | 'bool'
        | 'string'
        | 'byte'
;
```

Figure 5.1: Example Xtext Grammar

There are a few new concepts introduced in this example. First `'type'` and `'is'` appear in single quotes. This means that they are keywords instead of rules. The generated plug-in will give the option to highlight keywords however the user desires. The second is the `ID` rule. Xtext has a number of built in rules that are usually convenient. The `ID` rule is usually used to describe names that might be used in a program. The rule is defined:

```
terminal ID :
    ('^')?('a'..'z'|'A'..'Z'|'_')
    ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

Any built in rule can be overridden by declaring it in your grammar file. There aren't any more concepts to introduce with this example, but the full grammar for the example is given in Figure 5.1. The final step in building the plug-in is to run one of the files included when we created a new Xtext project. The working result is in Figure 5.2.

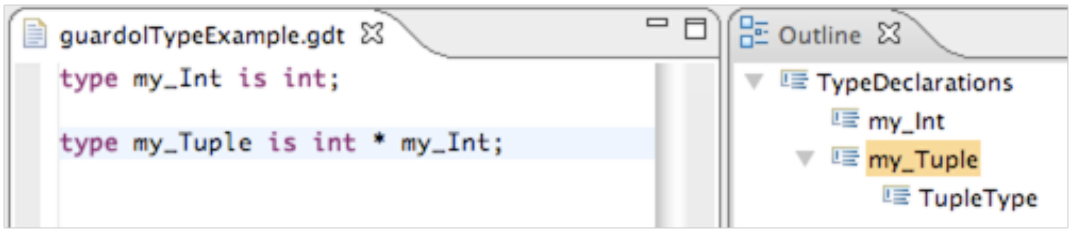

Figure 5.2: Xtext example plug-in

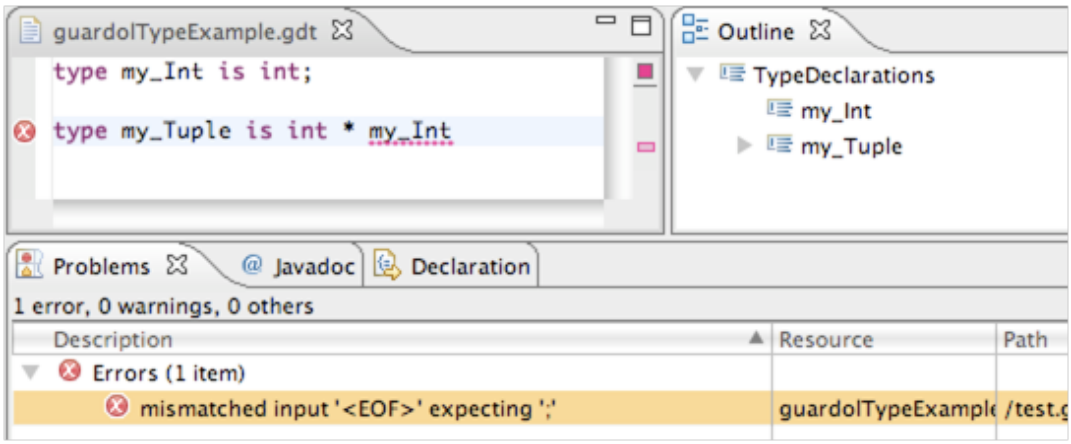

Figure 5.3: Example of Xtext Error display

5.1.2 Xtext Features

Figure 5.2 displays a few of the features that Xtext provides from only the grammar we built above. The first thing you can see is the syntax highlighting on the keywords. The words `type` and `is` are highlighted just as we expected. You can also see the collabsable outline view on the right side of the image. The outline view can be further tweaked by adding label icons and limiting the amount of information that is displayed. The outline can also be improved by adding additional information to the existing labels.

Another feature that Xtext provides out of the box is syntax error highlighting. If the file that is given doesn't parse correctly, Xtext will report it with a red underline and an error message in the problems view. An example of error highlighting can be seen in Figure 5.3

We can still do better if we put more work into it though. Figure 5.4 shows a case where we expect an error because `other_type` isn't defined anywhere in the code. To discover problems like this, Xtext allows you to define crosslinks in the AST. Figures 5.5 and reffig:Xtextast2 show how the AST might look before and after crosslinks are added.

It is easy to let Xtext know that we want a crosslink. The code responsible for resolving other_type in Figure 5.4 is SimpleType's ID option.

```
SimpleType :
    ID | "int" | "bool" | "byte" | "string";
```

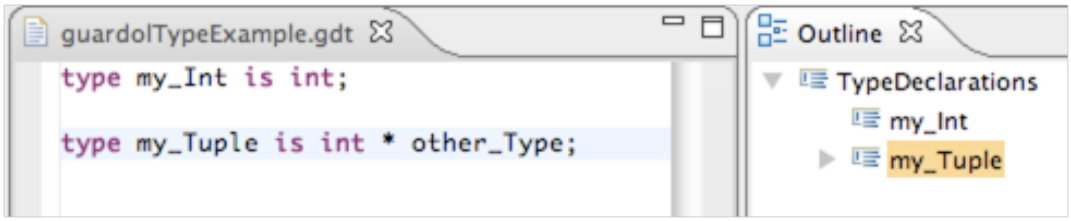

Figure 5.4: This Example Could Display another Error

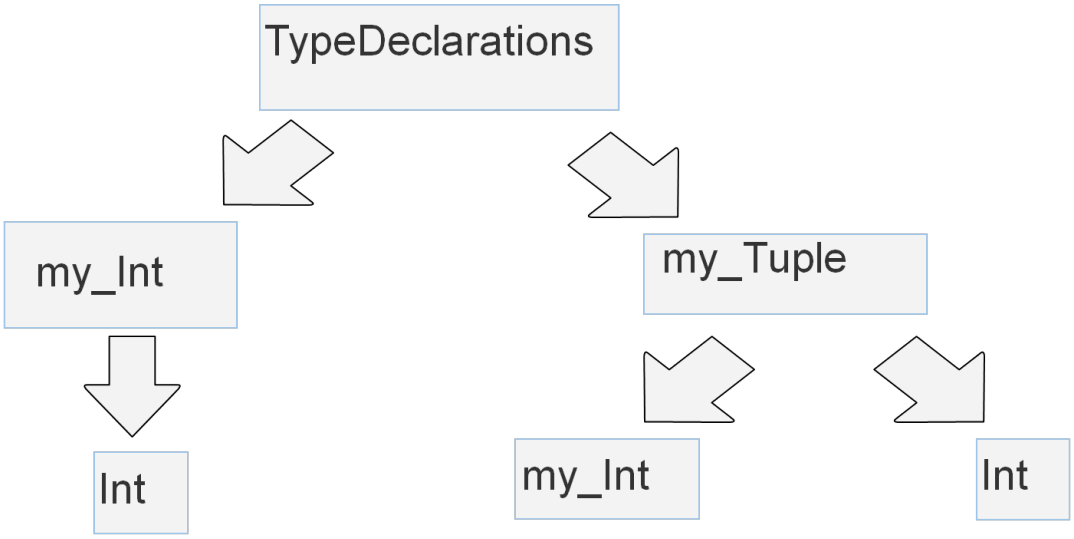

Figure 5.5: Xtext AST Before Crosslinks

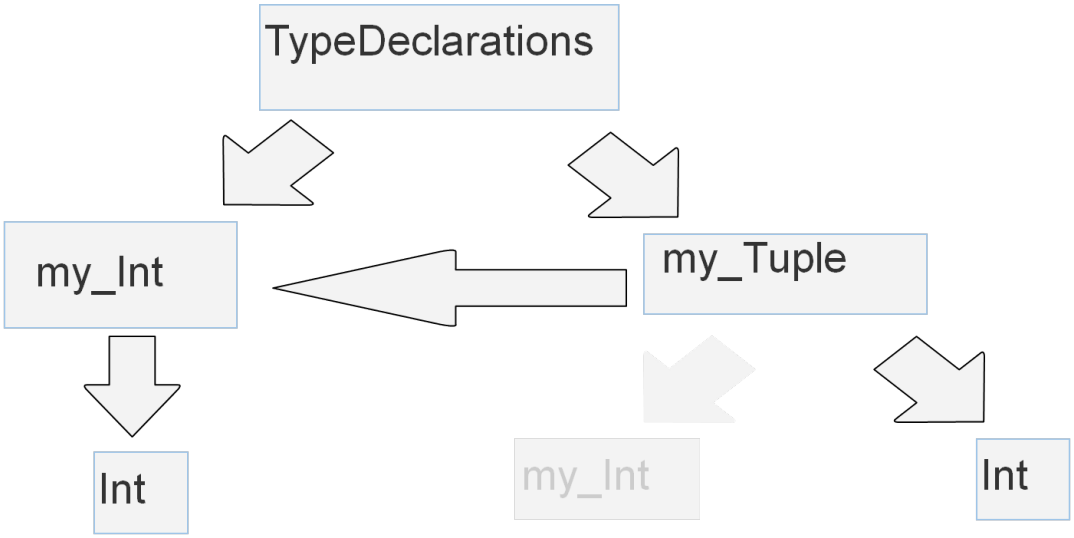

Figure 5.6: Xtext AST After Crosslinks

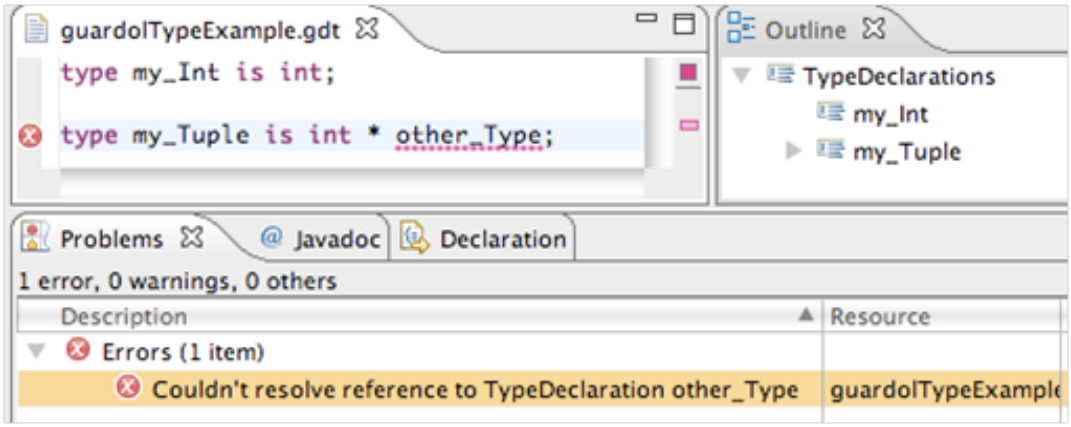

Figure 5.7: We now see an error thanks to crosslinks

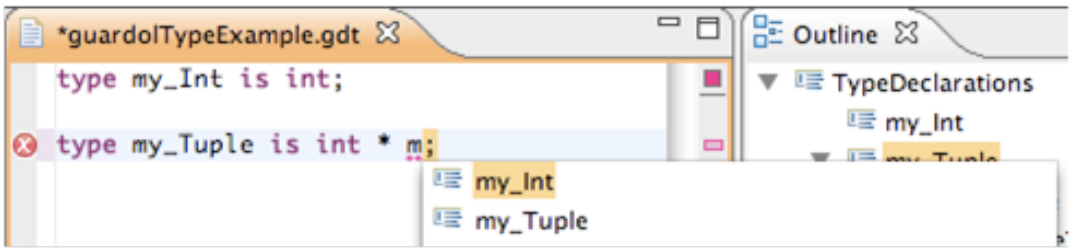

Figure 5.8: Xtext code suggestion

Instead of just using an ID, we want to link to an already existing type declaration. All that we have to do is change ID to [Typedeclaration] and we have a grammar capable of creating crosslinks. The updated rule looks like

```
SimpleType :
    dec=[TypeDeclaration] | "int" | "bool" | "byte" | "string";
```

Xtext's default behavior is to use the name field to resolve crosslinks which works fine in this case because TypeDeclaration uses a field called name to describe itself. Figure 5.7 now shows an error under other_type. We can also get some auto-complete options shown in figure 5.8.

5.2 Sireum plug-in

The rest of the Guardol plug-in is generated by the Sireum framework. The work done for the Sireum SPARK project was easy to convert to Guardol use. The two remaining features that we implemented with this plug-in were automatic builds and error highlighting for errors returned by any of our formal analyses. Automatic builds find .gdl file in a project that has the guardol nature and run the compiler on them. Running the compiler results in translation to XML, Pilar, and Simp. Figure 5.9 shows the output folder in an eclipse project. The automatic build was as simple as adding the name of our compile method

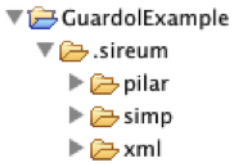

Figure 5.9: Output generated by the Sireum plugin

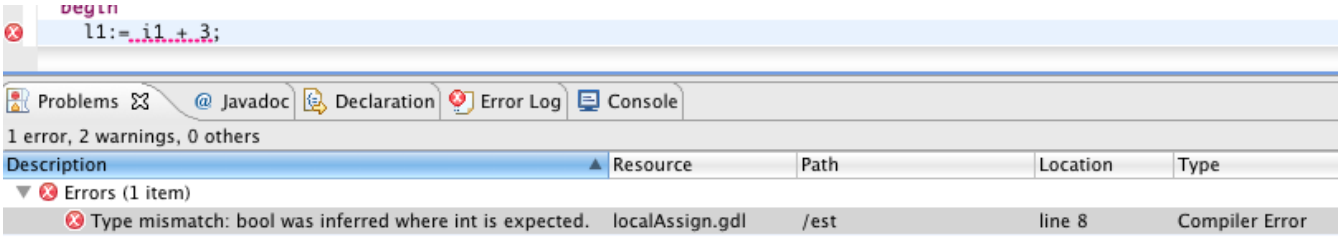

Figure 5.10: Error generated by the Sireum plugin

to the plug-in generator with the correct parameters. The plug-in adds a menu option to projects to add a Guardol nature. Any project with the Guardol nature will automatically build any files ending int the ".gdl" extension. The location of the output files can be configures by the user.

The second part of the plug-in visually reports errors to the user with a red underline. Once again the functionality was already written into the Sireum plug-in generator, all we had to do was pass it our error messages in the correct format. We were already using the Sireum message format which is a java class with a message and start and end line and column numbers. The start and end column and line numbers are used to create a selection that will receive a red underline for the error. The message is used to describe the error when the region is moused over. The message is also displayed in the eclipse problems view. Simply passing a list of these errors to the correct part of the plug-in generator was sufficient to highlight the errors in Guardol files. Figure 5.10 shows an error message generated by the Sireum plug-in.

Chapter 6

Core Language

This chapter presents a *core language* that represents the Guardol language. We use this language in order to describe our analyses on the Guardol language in a concise and accurate manner. It is important that every statement and expression available in Guardol is represented in some way by the core language. The list of statements in the core language are in figure 6.1 and the expressions are in Figure 6.2.

6.1 Statements

The core language statements appear very different from the Guardol statements, but it is possible to construct any of the Guardol statements from the core language statements. A summary of the statements follows:

- `assert`(B): The program only continues execution after this command if B evaluates to true. If B evaluates to false, execution aborts, generally with a message.

C	::=	`assert`(B)	// checks that B follows from current assumptions
	|	`assume`(B)	// adds B to current assumptions
	|	$x := E$	// identifier binding
	|	`local` x `in` C	// introduce local identifier
	|	$C_1 \,;\, C_2$	// first C_1 then C_2
	|	$C_1 \oplus C_2$	// either C_1 or C_2
	|	$C_1 \parallel C_2$	// run in parallel
	|	$x := n(y)$	// calling node

Figure 6.1: Statements in the core language

- `assume`(B): If B evaluates to false execution "dies silently", otherwise execution continues. `assume` is generally used to prune execution paths during verification.
- $x := E$: The value of expression E is bound to the identifier x.
- $C_1 \oplus C_2$: A *Non-deterministic choice* executes either C_1 or C_2.
- $C_1 \,;\, C_2$: *Sequential composition* executes C_1 followed by C_2.
- $x := n(y)$: *Node calls* apply the node n to the argument y and the result is bound to x. This can be generalized to multiple inputs.
- `local` x `in` C: *Local Identifiers* allows a node body to introduce identifiers in addition to the input and output identifiers.

From these definitions we can build every command in the Guardol language:

- Local variable declarations, Assignment, and sequential composition statements are all directly represented.
- If statements are represented by `assume` and *Non-deterministic choice*. For example the Guardol statement "if B then C_1 else C_2" would correspond to

  $$\texttt{assume}(B); C_1 \;\oplus\; \texttt{assume}(\neg B); C_2$$

  The idea here is that we can make a non-deterministic choice between every branch of the if statement. We then kill off every branch where the condition evaluates to false.
- Match statements are represented in a similar manner to if statements, but they require local declarations when the match declares names.
- Node calls are expressions in Guardol. We are able to elevate them to the level of commands in the core language because our IR uses three address code. It is acceptable to base our core language on our IR because we run all of our analyses on the IR.

6.2 Expressions

Every Guardol expression with the exception of the node call (see above) is directly represented by an expression in the core language.

We can write inference rules for a judgement $\rho \vdash E \Downarrow v$ that says that E evaluates to v in environment ρ that is functional and consistent. The rules are in figures 6.3 and 6.4.

To make the rules easier to understand we will walk through a few of them. The binary op rule looks like:

E	::=	x	// identifier
	|	c	// constant (natural number, boolean, etc.)
	|	$E : t$	// forces E to have type t
	|	$E_1\ op\ E_2$	// binary operation
	|	$(E_1, \ldots, E_k)$	// construct a tuple
	|	$\{f_1 : E_1, \ldots, f_k : E_k\}$	// construct a record
	|	$C\ E$	// construct a variant of a union type
	|	$E\#i$	// the i'th element of a tuple
	|	$E.f$	// the value of field f in a record
	|	E_1 `when` E_2	// returns E_1 if E_2 holds
	|	E_1 `default` E_2	// tries first E_1 next E_2
	|	`exists` E	// has E been assigned a value?

Figure 6.2: Core Language Expressions

$$\frac{\rho \vdash E_1 \Downarrow v_1 \qquad \rho \vdash E_2 \Downarrow v_2 \qquad v_1 \neq \textbf{ud} \qquad v_2 \neq \textbf{ud} \qquad v_3 = v_1\ op\ v_2}{\rho \vdash E_1\ op\ E_2 \Downarrow v_3}$$

The bottom half is straightforward. The assumptions on the top half of the form $\rho \vdash E_1 \Downarrow v_1$ mean that we will evaluate E_1 in our environment and call the result v_1. This can basically be seen as a recursive call to our rules. We then perform the binary operation and call the result v_3. v_3 is then the result ov evaluating a binary operation. We will now discuss the semantics that are the most unique to guardol.

We will first examine the semantics for the `when` operator. We can see that there are two different rules for the when operation. The first rule has the assumption that $\rho \vdash E_2 \Downarrow \mathsf{true}$. In this case we can see that the result of our evalutaion is the result of the evaluation of E_1 which is v. The second rule for `when` assums that $\rho \vdash E_2 \Downarrow \mathsf{true}$ is either false or **ud**. In this case the value of the expression is always **ud**.

Next we explain the rules for the `default` expression. The first case assums that the value v of $\rho \vdash E_1 \Downarrow v$ is not **ud**. In this case the value of the expression is v. The second rule covers the case $\rho \vdash E_1 \Downarrow \textbf{ud}$. In this case the value of the expression is the value of the evaluation of E_2.

Finally will dicsuss the `exists` rules. The first rule assumes that the value of E is not **ud**. In this case the value of the expression is true. If we assume that $\rho \vdash E \Downarrow \textbf{ud}$ the value of the `exists` expression is false.

$$\frac{\rho(x) = v}{\rho \vdash x \Downarrow v}$$

$$\frac{}{\rho \vdash c \Downarrow c}$$

$$\frac{\rho \vdash E_1 \Downarrow v_1 \qquad \rho \vdash E_2 \Downarrow v_2 \qquad v_1 \neq \mathbf{ud} \qquad v_2 \neq \mathbf{ud} \qquad v_3 = v_1 \; op \; v_2}{\rho \vdash E_1 \; op \; E_2 \Downarrow v_3}$$

$$\frac{\rho \vdash E_1 \Downarrow v_1 \qquad \rho \vdash E_2 \Downarrow v_2 \qquad v_1 = \mathbf{ud}}{\rho \vdash E_1 \; op \; E_2 \Downarrow \mathbf{ud}}$$

$$\frac{\rho \vdash E_1 \Downarrow v_1 \qquad \rho \vdash E_2 \Downarrow v_2 \qquad v_2 = \mathbf{ud}}{\rho \vdash E_1 \; op \; E_2 \Downarrow \mathbf{ud}}$$

$$\frac{\rho \vdash E_1 \ldots E_k \Downarrow v_1 \ldots v_k \qquad \{v_1, \ldots, v_k\} \cap \{\mathbf{ud}\} = \emptyset}{\rho \vdash (E_1, \ldots, E_k) \Downarrow v}$$

$$\frac{\rho \vdash E_1 \ldots E_k \Downarrow v_1 \ldots v_k \qquad \{v_1, \ldots, v_k\} \cap \{\mathbf{ud}\} \neq \emptyset}{\rho \vdash (E_1, \ldots, E_k) \Downarrow \mathbf{ud}}$$

$$\frac{\rho \vdash E_1 \ldots E_k \Downarrow v_1 \ldots v_k \qquad \{v_1, \ldots, v_k\} \cap \{\mathbf{ud}\} = \emptyset}{\rho \vdash \{f_1 : E_1, \ldots, f_k : E_k\} \Downarrow v}$$

$$\frac{\rho \vdash E_1 \ldots E_k \Downarrow v_1 \ldots v_k \qquad \{v_1, \ldots, v_k\} \cap \{\mathbf{ud}\} \neq \emptyset}{\rho \vdash \{f_1 : E_1, \ldots, f_k : E_k\} \Downarrow \mathbf{ud}}$$

$$\frac{\rho \vdash E \Downarrow v_1 \qquad v_1 \neq \mathbf{ud} \qquad v = C \; v_1}{\rho \vdash C \; E \Downarrow v}$$

$$\frac{\rho \vdash E \Downarrow v_1 \qquad v_1 = \mathbf{ud}}{\rho \vdash C \; E \Downarrow \mathbf{ud}}$$

$$\frac{\rho \vdash E \Downarrow v_1 \qquad v_1 \neq \mathbf{ud} \qquad v = v_1 \# i}{\rho \vdash E \# i \Downarrow v}$$

$$\frac{\rho \vdash E \Downarrow v_1 \qquad v_1 = \mathbf{ud}}{\rho \vdash E \# i \Downarrow \mathbf{ud}}$$

$$\frac{\rho \vdash E \Downarrow v_1 \qquad v_1 \neq \mathbf{ud} \qquad v = v_1.f}{\rho \vdash E.f \Downarrow v}$$

$$\frac{\rho \vdash E \Downarrow v_1 \qquad v_1 = \mathbf{ud}}{\rho \vdash E.f \Downarrow \mathbf{ud}}$$

Figure 6.3: Value Inferences on Core Language Expressions

$$\frac{\rho \vdash E_2 \Downarrow \mathsf{true} \qquad \rho \vdash E_1 \Downarrow v}{\rho \vdash E_1 \texttt{ when } E_2 \Downarrow v}$$

$$\frac{\rho \vdash E_2 \Downarrow v \qquad v \in \{\mathsf{false}, \mathbf{ud}\}}{\rho \vdash E_1 \texttt{ when } E_2 \Downarrow \mathbf{ud}}$$

$$\frac{\rho \vdash E_1 \Downarrow v_1 \qquad v_1 \neq \mathbf{ud}}{\rho \vdash E_1 \texttt{ default } E_2 \Downarrow v_1}$$

$$\frac{\rho \vdash E_1 \Downarrow \mathbf{ud} \qquad \rho \vdash E_2 \Downarrow v}{\rho \vdash E_1 \texttt{ default } E_2 \Downarrow v}$$

$$\frac{\rho \vdash E \Downarrow v \qquad v \neq \mathbf{ud}}{\rho \vdash \texttt{exists } E \Downarrow \mathsf{true}}$$

$$\frac{\rho \vdash E \Downarrow \mathbf{ud}}{\rho \vdash \texttt{exists } E \Downarrow \mathsf{false}}$$

Figure 6.4: Value Inferences on Core Language Expressions Continued

Chapter 7

Formal Analyses

7.1 Problem Formulation

As explained in Chapter 6, Guardol variables may be classified as either partial or total. To be considered total a variable must

1. Always be defined when it is referenced
2. Always be defined at the end of the node it is declared in

Guardol also imposes rules requiring that no variable can be assigned more than once. In this chapter we define static analyses for enforcing the constraints on partial and total variables, as well as enforcing that no variables are assigned more than once.

We will review some terminology to make these concepts more precise. A *proper* value is one that is not **ud**. When executing a command $x := E$, we say that x is *assigned* (even if E evaluates to **ud**), and the variables in E are *referenced*. Variables may also be referenced through other constructs.

We are given the following *requirements* on the execution of a Guardol program. On any execution path through a node

1. [Single Assignedness] a variable cannot be assigned more than once - We will refer to this as the assignedness property
2. [Total Variable Definedness] Whenever a *total* variable is referenced, it must already have a proper value - This is the definedness property

It is convenient to assume that a total variable is implicitly referenced at the end of each method body. Then our requirements imply that along each execution path

> a total variable is assigned exactly once, at which time it is given a proper value.

There are no restrictions on the definedness of a partial variable but we shall issue a *warning* if a partial variable is assigned nowhere in the program. (But it is OK if it is assigned along some path but not assigned along some other path.)

We shall pursue a 2-phase static analysis approach to allow us to decide if a variable can be total.

1. Phase one is an "assignment analysis" to determine

   aa(a) the set of variables that are assigned exactly once in every possible execution,

   aa(b) the set of variables that are never assigned in any execution, and

   aa(c) the set of variables that are referenced before they are assigned along at least one path through the program.

   Variables that belong to aa(a) are candidates for being declared total. As a side effect we issue an error if

   e(a) any variable is assigned more than once,

   e(b) a variable is declared total but does not appear in the set of variables that are always assigned aa(c), or

   e(c) a variable is declared total and appears in the set of variables referenced before they are assigned aa(c).

   The analysis will issue a warning if any variable appears in the set of variables that are never assigned.

   At this point the analysis knows that a total variable has always been assigned but we don't know if it has always been given a proper value. The second phase enforces that constraint.

2. Phase two is a "definedness analysis" that discovers variables that are always given a proper value. This analysis generates

   (a) a set of constraints that *might* require variables to be partial and

   (b) a set of constraints that always require certain variables to be total

   the constraints can then be solved. In the case that there is no solution to the constraints because of a conflict between (a) and (b), the analysis will generate an error.

   When this analysis completes, we will be able to determine which variables are always given a proper value.

A variable can be total if it belongs to the set of variables that are always assigned exactly once aa(a) and the set of variables that are always given a proper value (2). If the program specifies any variables as total and they don't belong to those two sets the analysis will report an error. If a variable is not declared total the analysis can still discover that it is safe to *consider* it total. It is safe for the analysis to make this assumption if a variable belongs to sets aa(a) and (2) as defined above. In this case the analysis can either issue a message to the user recommending that they change the declaration. It can also proceed with analysis under the assumption that the variable is total.

We cannot guarantee complete accuracy because of the nature of static analysis. Therefore, we design the analysis in a conservative matter to guarantee the following soundness property:

> If the analysis classifies a variable as total, its runtime properties satisfy the [Total Variable Definedness] propery.

There may be variables, whoever, whose runtime behavior *does* satisfy [Total Variable Definedness] even though the analysis fails to classify them as total. So when viewing the analysis as an oracle that reports if there is an error in the declaration of total variables, the analysis may produce "false positives".

7.2 Assignment Analysis

The first step in the enforcement of our rules is the assignedness analysis. The goal of the assigned analysis is to determine if on any path through the CFG a variable might not be assigned, if it will always be assigned, or if there is no chance that it will be assigned.

We use the domain $\mathcal{A} = \{0, 1, ?\}$ where 0 means "definitely not assigned"; 1 means "definitely assigned"; ? means "may or may not be assigned". We let a range over values in that domain which we equip with a partial order $\sqsubseteq$ by stipulating $0 \sqsubseteq ?$ and $1 \sqsubseteq ?$. We let $a_1 \sqcup a_2$ denote the least upper bound wrt. that ordering: $0 \sqcup 0 = 0$, $1 \sqcup 1 = 1$, and otherwise $a_1 \sqcup a_2 = ?$.

We let A range over "assignedness environments", that is, mappings from variables to $\mathcal{A}$. For example $A = [x \mapsto 0, y \mapsto 1, z \mapsto ?]$ is an environment that classifies x as as definitely not assigned, y as always assigned and z as unknown. We extend $\sqcup$ pointwise to assignedness environments, and use $\vec{0}$ to denote the environment that maps all variables to 0.

The analysis described so far is sufficient to compute sets aa(a) and aa(b) above. To determine aa(c) we add a component Γ representing the set of variables that are referenced before they are assigned on at least one branch of the program. Variables contained in Γ can no longer be considered possible total variables. Γ is independent of A, so a variable in Γ can map to any value in $\mathcal{A}$. The only way to combine two Γs is with the $\cup$ operation. As a result the functions for combining Γs are monotonic; Variables are only added to Γ and never removed. This is consistent with our description in aa(c) above. We are interested in variables that are referenced before assigned *on at least one path through the program.*

We also need two operations to determine if a variable is referenced before it assigned. First, we define $\mathsf{Ref}(E)$ which collects all of the variables that are referenced in an expression. Using that definition we define $\mathsf{RefBeforeAssign}_A(E)$ which checks for variables that are referenced before assigned. $\mathsf{RefBeforeAssign}_A(E) = \mathsf{Ref}(E) \cap \{x | A(x) \in \{0, ?\}\}$. This definition is used to determine which variables should be inserted into Γ.

Figure 7.1 presents an inference system with judgements of the form $\langle A,\ \Gamma\rangle \vdash C : \langle A',\ \Gamma'\rangle$. This means that if A and describes assignedness *before* executing command C then A' describes what is assigned directly after C. Similarly Γ is the set described by aa(c) before command C and Γ' is the updated set after C.

$$\frac{}{\langle A,\ \Gamma\rangle \vdash \texttt{assume}(E) : \langle A,\ \Gamma \cup \mathsf{RefBeforeAssign}_A(E)\rangle}$$

$$\frac{}{\langle A,\ \Gamma\rangle \vdash \texttt{assert}(E) : \langle A,\ \Gamma \cup \mathsf{RefBeforeAssign}_A(E)\rangle}$$

$$\frac{}{\langle A,\ \Gamma\rangle \vdash \texttt{local}\ x\ \texttt{in}\ C : \langle A[x \mapsto 0],\ \Gamma\rangle}$$

$$\frac{\langle A,\ \Gamma\rangle \vdash C_1 : \langle A',\ \Gamma'\rangle \qquad \langle A',\ \Gamma'\rangle \vdash C_2 : \langle A'',\ \Gamma''\rangle}{\langle A,\ \Gamma\rangle \vdash C_1\,;\, C_2 : \langle A'',\ \Gamma''\rangle}$$

$$\frac{\langle A,\ \Gamma\rangle \vdash C_1 : \langle A_1,\ \Gamma_1\rangle \qquad \langle A,\ \Gamma\rangle \vdash C_2 : \langle A_2,\ \Gamma_2\rangle}{\langle A,\ \Gamma\rangle \vdash C_1 \oplus C_2 : \langle A_1 \sqcup A_2,\ \Gamma_1 \cup \Gamma_2\rangle}$$

$$\frac{A(x) = 0}{\langle A,\ \Gamma\rangle \vdash x := n(y) : \langle A[x \mapsto 1],\ \Gamma \cup \mathsf{RefBeforeAssign}_A(y)\rangle} \quad // \text{ if } A(x) \neq 0 \text{ then ERROR}$$

$$\frac{A(x) = 0}{\langle A,\ \Gamma\rangle \vdash x := E : \langle A[x \mapsto 1],\ \Gamma \cup \mathsf{RefBeforeAssign}_A(E)\rangle} \quad // \text{ if } A(x) \neq 0 \text{ then ERROR}$$

Figure 7.1: Selected rules for $\langle A,\ \Gamma\rangle \vdash C : \langle A',\ \Gamma'\rangle$.

The rules from Figure 7.1 describe the intra-node aspects of the analysis. To prepare for analyzing a node, the structures A and Γ must be properly initialized. A is initialized by adding variables that are listed as inputs to A. We do this because the inputs to the node are assigned before the node begins execution. The only way to distinguish the assignedness status of input variables is to check if they are declared total. If they are declared total, we know that they must always be assigned and we use the command $A[x \mapsto 1]$ where x is an input variable. We cannot determine the assignedness of any input variables that are not declared total so we use the command $A[x \mapsto ?]$. When initializing A it is also convenient to create a set T of all variables that are declared total in the node. Γ is initialized to the empty set. To begin the algorithm we perform $\langle A,\ \Gamma\rangle \vdash C : \langle A',\ \Gamma'\rangle$ where C is the body of the node. Γ is initialized as $\emptyset$.

If the algorithm completes successfully we have A' and Γ'. The analysis will have created errors if it found any duplicate assignments. We now need to check our results against the total annotated variables we stored in T. If $T - A.aa \neq \emptyset$ we return an error stating that the set of variables $T - A.aa$ are annotated total but may not be assigned on some program execution. We then check $T \cap \Gamma = \emptyset$. If that is false we generate an error stating that the set of variables $T \cap \Gamma$ are declared total, but could be used before they are assigned on some program execution.

We have now shown that our requirements on assignedness are held, and we can procede to generating constraints on definedness.

An alternate view of A is as a pair of sets containing all of the variables that are always

assigned (aa) and all of the variables that are never assigned (na). This is the view used in the implementation of the static analysis. These sets contain all variables that map to 1 and 0 respectively. Any variable that is not in one of these sets can be assumed to be mapped to the set of all variables that may or may not be assigned (uk). If we keep an environment V containing the names of all local variables in a node we can easily derive uk from A and aa. The function for this derivation is $A_V.uk = V - (A.aa \cup A.na)$. The calculation of V only requires an examination of the node signature and the declared local variables. This calculation is trivial enough that we will not explicitly refer to V even though it is implied in all references to A. We can define a function $A(x)$ where $A(x)$ returns 1 if $x \in aa$, 0 if $x \in na$, or ? otherwise. We can also extend the $A_1 \sqcup A_2$ operation to this environment. We will refer to the result of $A_1 \sqcup A_2$ as A_3. $A_3.aa$ will simply be $A_1.aa \cap A_2.aa$. This is equivalent to the extension of $\sqcup$ above because for $A_3.aa$, $A(x) = 1$ if and only if $A_1(x) = 1$ `and` $A_2(x) = 1$. $A_3.na$ will similarly be defined as $A_1.aa \cap A_2.aa$. $A_3.uk$ can now be derived according to the above function for deriving uk. It is also convenient to declare an operation $A[x] = A.aa \cup \{x\}$ that adds a variable to A's aa set and removes it from the na set.

7.3 Definedness Analysis

We use a two-point domain $\mathcal{D} = \{\mathsf{T}, \mathsf{P}\}$ where T (total) means "definitely defined" and P (partial) means "may or may not be defined" (we have no use for "definitely not defined"). We let d range over values in $\mathcal{D}$ which we equip with a partial order $\sqsubseteq$ by stipulating $\mathsf{T} \sqsubseteq \mathsf{P}$. We let $d_1 \sqcup d_2$ denote the least upper bound wrt. that ordering: $\mathsf{T} \sqcup \mathsf{T} = \mathsf{T}$, and otherwise $d_1 \sqcup d_2 = \mathsf{P}$. Dually, we define $d1 \sqcap d2$ as the greatest lower bound where $\mathsf{P} \sqcup \mathsf{P} = \mathsf{P}$, and otherwise $d_1 \sqcup d_2 = \mathsf{T}$. A *definedness environment* Δ maps each *expression*, in particular each variable, into $\mathcal{D}$.

We use DC to range over sets of *definedness constraints* which are of the form $dl \sqsubseteq \Delta(E)$ where each dl is of the form either $\Delta(E_0)$, P, or $\Delta(E_1) \sqcap \Delta(E_2)$. Because each such left hand side is monotone in Δ, we infer that

> Given a set of definedness constraints DC, there exists a *least* definedness environment Δ that satisfies DC.

This means that if we pass any Guardol program through our analysis some set of definedness constraints will be generated.

We will present the definedness rules in two different forms. The first form is a list of type rules, which give a process for determining if any given expression is partial or total. The second method is rules for generating constraints. The first method is generally easier to read because it is directly determining types instead of generating constraints that can be used to determine types. The constraint generation rules are presented because the compiler's partial total analysis algorithm is derived from them.

Figure 7.3 presents an inference system with judgements of the form $\Sigma \vdash e : T$. This means given the environment the expression e will be judged to have type T. Now we will discuss a few of the rules. The first rule defines the behavior when we see a variable. In this

```
node n (b : int) returns
(o : int total, z: int total, x: int)
is
begin
if b then
   o := 3;
else
   o := 3 when false;
end if
   z := o;
   x := 3;
```

Figure 7.2: Constraint generation example

case, we will check to see if the variable is total or partial in our environment. When we look at these rules, we need to understand that their purpose is to describe an environment that is a correct solution to the constraints we generate. That means that once we have generated and solved our constraints, we will have an envirnoment. A correct environment will allow for the rules to be applied correctly. We also remember that there might be a number of correct environments for any given program. Our goal in general is to find the environment that gives us as many total variables as possible. Most of the rest of the rules follow directly from the value inference rules from the core language (Figure 6.3). `when` is always partial, for example because we can't decide the boolean variable in a static analysis.

We write a function Gen that generates constraints from

- a command C or an expression E;

Key clauses can be found in Fig. 7.4. This function is derived from the inference rules in Figure 7.3.

Gen generates a number of constraints. Each will constrain some expression either to P or to P if some other expression is P. The first two rules are for variables and constants. In these cases we will not generate any constraints because nothing in these expressions could force a variable to be partial. When we examine the rule for a binary operation we can see the constraints that are generated. As described in the semantics, if either E_1 of E_2 is undefined then E should also be undefined. To match this, we generate two constraints so that E is P if either E_1 or E_2 is P. The rest of the rules follow similarly from the semantics and the rules from Figure 7.3. Because we only add constraints, in all cases they are unioned into a set of constraints. Using the sequece rule, we can build a complete list of constraints for any program.

For an example, we will discuss the constraints that will be generated when the function is run on the code in Figure 7.2.

To start, we examine the node signature. We add totality constraints for o and z because they are declared total in the node signature. We don't do anything to b because at this point it could be total, but it doesn't have to be. We move on to the if statement where we see that the variable b is used as the boolean expression of an if statement. By our rules,

we also add a totality constraint for b. Next we examine the true branch of the if. Here we add an assignment constraint that if $\Delta(3)$ is P then $\Delta(o)$ must also be P. The constraint that represents this is $\Delta(3) \sqsubseteq \Delta(o)$. This constraint clearly isn't useful, but we will discover that when we try to find a solution.

Now we examine the next assignment, where we have to generate two different constraints. The first constraint is for the expression $3 when false$. By the rule for when we get the constraint $\mathsf{P} \sqsubseteq \Delta(3 when false)$. We also generate $\Delta(3 when false) \sqsubseteq \Delta(o)$ using the assignment rule. This constraint exposes that Δ is not a mapping from variables to $\mathcal{D}$, but a mapping from expressions to $\mathcal{D}$. In this use, the expression isn't evaluated, simply used as a label to find the associated value in $\mathcal{D}$.

We only need to use the assignment rules for the next two statements. When we complete the constraint generation we have totality constraints o, z, b. We also have the following constraints:

1. $\Delta(3) \sqsubseteq \Delta(o)$
2. $\mathsf{P} \sqsubseteq \Delta(3 when false)$
3. $\Delta(3 when false) \sqsubseteq \Delta(o)$
4. $\Delta(o) \sqsubseteq \Delta(z)$
5. $\Delta(3) \sqsubseteq \Delta(x)$

7.4 Constraint Solving

To solve our constraints we initialize a Δ by performing $\Delta[x \mapsto \mathsf{T}]$ for all variables in the program. The algorithm then iterates over DC. For each constraint, it retrieves the mapping in Δ for the left hand side. If the left hand side is T the algorithm continues to iterate. If the left side is P, however the algorithm uses the command $\Delta[x \mapsto \mathsf{P}]$ where the constraint was $\mathsf{P} \sqsubseteq \Delta(x)$. This will overwrite the previous mapping for x in Δ. For efficiency we can now choose to remove the current constraint from DC before the algorithm moves to the next constraint. We can do this because we know the constraints are monotone in Δ which means no constraint will be able to modify Δ twice. While the algorithm is iterating over DC it remembers if it has seen any left hand sides that evaluate to P. If it completes an entire iteration without seeing this case, it has found Δ that is a least solution to DC.

In addition, we use a TC to range over *totality constraints*, of the form $\Delta(E) = \mathsf{T}$. Even though the resulting Δ is a solution to DC it may not be an acceptable solution for the program. The algorithm now must check for conflicts with TC. It does this with a single iteration over TC. For every constraint $\Delta(E) = \mathsf{T}$ it checks that $DC(E) = \mathsf{T}$. If $DC(E) = \mathsf{P}$ there is no solution to the constraints and the algorithm issues an error. If Δ is compatible with every constraint in TC then Δ is a solution for the program.

We will now find a solution to the constraints that we generated in our previous example. The guardol code for the example is in Figure 7.2 and the constraints are

$$\frac{}{\Sigma \vdash\ x\ : \Sigma(x)}$$

$$\frac{}{\Sigma \vdash\ c\ : \mathsf{Total}}$$

$$\frac{\Sigma \vdash\ E_1\ : T_1 \qquad \Sigma \vdash\ E_2\ : T_2 \qquad T_3 = T_1 \sqcup T_2}{\Sigma \vdash\ E_1\ op\ E_2\ : T_3}$$

$$\frac{\Sigma \vdash\ E_{1\ldots k}\ : T_{1\ldots k} \qquad T = T_1 \sqcup \ldots \sqcup T_k}{\Sigma \vdash\ (E_1, \ldots, E_k)\ : T}$$

$$\frac{\Sigma \vdash\ E_{1\ldots k}\ : T_{1\ldots k} \qquad T = T_1 \sqcup \ldots \sqcup T_k}{\Sigma \vdash\ \{f_1 : E_1, \ldots, f_k : E_k\}\ : T}$$

$$\frac{\Sigma \vdash\ E\ : T}{\Sigma \vdash\ CE\ : T}$$

$$\frac{\Sigma \vdash\ E\ : T}{\Sigma \vdash\ E\#i\ : T}$$

$$\frac{\Sigma \vdash\ E\ : T}{\Sigma \vdash\ E.f\ : T}$$

$$\frac{\Sigma \vdash\ E_1\ : T_1 \qquad \Sigma \vdash\ E_2\ : T_2}{\Sigma \vdash\ E_1\ \texttt{when}\ E_2\ : \mathsf{Partial}}$$

$$\frac{\Sigma \vdash\ E_1\ : T_1 \qquad \Sigma \vdash\ E_2\ : T_2 \qquad T_3 = T_1 \sqcap T_2}{\Sigma \vdash\ E_1\ \texttt{default}\ E_2\ : T_3}$$

$$\frac{\Sigma \vdash\ E\ : T}{\Sigma \vdash\ \texttt{exists}\ E\ : \mathsf{Total}}$$

$$\frac{\Sigma \vdash\ E\ : \mathsf{Total}}{\Sigma \vdash\ \texttt{assert}(E)\ :}$$

$$\frac{\Sigma \vdash\ E\ : \mathsf{Total}}{\Sigma \vdash\ \texttt{assume}(E)\ :}$$

$$\frac{\Sigma \vdash\ E\ : T \qquad \Sigma(E) \sqsubseteq \Sigma(x)}{\Sigma \vdash\ x := E\ :}$$

Figure 7.3: Totality rules $\Sigma \vdash\ E\ : t$

$$\frac{}{x : \emptyset}$$

$$\frac{}{c : \emptyset}$$

$$\frac{E = E_1 \; op \; E_2 \qquad E_1 : DC_1 \qquad E_2 : DC_2}{E : \{\Delta(E_1) \sqsubseteq \Delta(E),\ \Delta(E_2) \sqsubseteq \Delta(E)\} \cup DC_1 \cup DC_2} \quad // \text{ if } E \text{ defined then both } E_1 \text{ and } E_2 \text{ defined}$$

$$\frac{E = \texttt{exists}\ E_1 \qquad E_1 : DC}{E : DC} \quad // \text{ no extra constraints as } E \text{ always defined}$$

$$\frac{E = E_1\ \texttt{when}\ E_2 \qquad E_1 : DC_1 \qquad E_2 : DC_2}{E : \{\mathsf{P} \sqsubseteq \Delta(E)\} \cup DC_1 \cup DC_2} \quad // \; E \text{ may be undefined}$$

$$\frac{E = E_1\ \texttt{default}\ E_2 \qquad E_1 : DC_1 \qquad E_2 : DC_2}{E : \{\Delta(E_1) \sqcap \Delta(E_2) \sqsubseteq \Delta(E)\} \cup DC_1 \cup DC_2} \quad // \text{ if } E \text{ defined then either } E_1 \text{ or } E_2 \text{ defined}$$

$$\frac{E : DC}{x := E : DC \cup \{\Delta(E) \sqsubseteq \Delta(x)\}} \quad // \; x \text{ may only be defined if } E \text{ is always defined}$$

$$\frac{E : DC}{\texttt{assume}(E) : DC}$$

$$\frac{E : DC}{\texttt{assert}(E) : DC}$$

$$\frac{C_1 : DC_1 \qquad C_2 : DC_2}{C_1 \,;\, C_2 : DC_1 \cup DC_2}$$

Figure 7.4: $\mathsf{Gen}_A(E) = DC$ holds iff $E : DC$ according to the rules above.

1. $\Delta(3) \sqsubseteq \Delta(o)$,
2. $\mathsf{P} \sqsubseteq \Delta(3 when false)$,
3. $\Delta(3 when false) \sqsubseteq \Delta(o)$,
4. $\Delta(o) \sqsubseteq \Delta(z)$, and
5. $\Delta(3) \sqsubseteq \Delta(x)$

.

We begin by creating an initial Δ that we will update as we proceed through our constraints. We initialise this Δ to the least solution we could have by viewing only the node signature. That means that all expressions map to T except for inputs that aren't declared total which must map to P. Now we iterate over constraints. For constraint 1 we first look up $\Delta(3)$ which is obviously T. If the left side of a constraint is T then nothing is constrained to P so we skip this rule and move on. The left hand side in this case is P so we modify our Δ so that $\Delta(3 when false) = \mathsf{P}$. Now due to the monotone property of our constraints we can remove the constraint from our list of constraint. This is because once we constrain an expression to partial, no constarint can move it back to total, so the constraint is no longer useful to us. The next step proceeds similarly constraining $\Delta(o)$ to partial. Step 4 constrains $\Delta(z)$ to P and step 5 doesn't change anything. We have now traversed all of our constraints, and updated Δ accordingly. We also removed all of the constraints that made changes to Δ. Now we perform another traversal, but we don't make any changes because each right hand side evaluates to T. This means that we have reached a fixed Δ that is an acceptable solution for our constraints.

Even though we have this Δ we don't have a correct solution to our program, because none exists. We now have to check our Δ against the totality constraints o, z, b. All three of these variables must be T but we constrained $\Delta(o)$ and $\Delta(z)$ to P. We must issue an error for these two variables.

Not as interesting in this example, but interesting in other cases is that x still maps to T even though it is not declared total in the node declaration. We can check its value in A. If the value is aa then in compilation or translations we can safely assume that x is annotated total. We can also issue a waning telling the programmer that x could be annotated total.

7.5 Mixed Tuples

We also use a component M which is a set of all expressions that are mixed tuples. Mixed tuples are tuples that are not strict which means they may have some partial and some total constraints. They can only be generated by calls to a node which does not return all total values. These tuples are not allowed to be bound to a name that is declared in the Guardol program. It is important to specify that the names are declared in the Guardol program because our analysis runs on our IR. Our IR uses three address code, which means it will introduce a number of temporary variables.

$$\frac{\langle \Gamma,\ M\rangle \vdash C_1 : M' \qquad \langle \Gamma,\ M'\rangle \vdash C_2 : M''}{\langle \Gamma,\ M\rangle \vdash C_1\,;\, C_2 : M''}$$

$$\frac{\langle \Gamma,\ M\rangle \vdash C_1 : M' \qquad \langle \Gamma,\ M'\rangle \vdash C_2 : M'}{\langle \Gamma,\ M\rangle \vdash C_1 \oplus C_2 : M''}$$

$$\frac{n \in M \qquad M' = M \cup \{x\}}{\langle \Gamma,\ M\rangle \vdash x := n(y) : M'} \quad \text{// if } x \in \Gamma \text{ then ERROR}$$

$$\frac{n \notin M}{\langle \Gamma,\ M\rangle \vdash x := n(y) : M}$$

$$\frac{\mathsf{Mixed}_M(E) = \mathsf{true} \qquad M' = M \cup \{x\}}{\langle \Gamma,\ M\rangle \vdash x := E : M'} \quad \text{// if } x \in \Gamma \text{ then ERROR}$$

$$\frac{\mathsf{Mixed}_M(E) = \mathsf{false}}{\langle \Gamma,\ M\rangle \vdash x := E : M}$$

Figure 7.5: Mixed Tuple rules for $\langle \Gamma,\ M\rangle \vdash E : M'$.

Because there are some variables that can have mixed tuples assigned to them we also keep a component Γ as a set of all Guardol variable names declared in a node.

The analysis consists of two parts. The first is a list of rules $\langle \Gamma,\ M\rangle \vdash E : M'$ that runs on commands and ensures that a mixed tuple is never assigned to a variable that was declared in the Guardol code. These rules appear in Figure 7.5. Certain commands are not included because they can't have expressions with mixed tuples. These rules simply return the same M and Γ as they were given. The second part is a function $\mathsf{Mixed}_M(E)$ which operates on an expression and returns a boolean value. It returns true if the value of the expression is a mixed tuple and false otherwise. There are a number of constructs that are not represented in the rules. We assume that if there is no rule we will run $\mathsf{Mixed}_M(E)$ for all sub expressions. If any of the results are true we will terminate with an error because mixed tuples are not allowed in these expressions. Otherwise we return a value of false. The rules defining this function are in figure 7.6

$$\frac{}{M \vdash\ x\ : x \in M}$$

$$\frac{}{M \vdash\ c\ : \mathsf{false}}$$

$$\frac{}{M \vdash\ E\#i\ : \mathsf{false}}$$

$$\frac{M \vdash\ E_1\ : B}{M \vdash\ E_1\ \texttt{when}\ E_2\ : B}$$

$$\frac{M \vdash\ E_1\ : B}{M \vdash\ E_1\ \texttt{default}\ E_2\ : B}$$

Figure 7.6: $\mathsf{Mixed}_M(E) = b$ holds if $E : b$ as defined above

Chapter 8

Simp Translation

The final step the compiler takes is to translate from the Pilar representation into Simp. To get an idea for how the translation works, we will walk through an example of the translation of the MIME example. The section of the MIME example we are working with is in Figure 8.1.

Both Simp and Pilar are ML like languages, which makes most of the translation somewhat simple. There are two issues, however, that make the translation difficult

1. Simp does not allow for undefined variables, but Guardol does and
2. the Pilar representation introduced temporary variables that may not have an explicit type

An example of the first issue is our type `MIME_Type`. In Guardol, the declaration of the type always allows for a variable of that type to be undefined. The Guardol type declaration is:

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

The corresponding declaration in Simp is very similar:

```
type MIME_Type = [body : string ; attachments : AttachmentList];
```

The problem with Simp, however, is that that type declaration will not allow for an undefined variable of the type `MIME_Type`. To allow for this type of variable we add a union declaration

```
type MIME_Type_option = { SOME : MIME_Type | NONE };
```

The name of they type is the same as the actual type with `_option` appended. The union simply consists of `SOME` of the original type or `NONE`. When we put a value into one of these "option unions" we refer to it as *wrapping* the value. When we take a value out we refer to it as *unwrapping*.

```
node MIME_Check
 (Input : MIME_Type total) 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;
```

Figure 8.1: Declaration of the *MIME_Check* node

The second problem relates to the first problem. Wrapping and unwrapping values in expressions can quickly get expensive and ugly, especially if the expressions are large. This is because we need to take different actions depending on weather each value in the expression is defined or not at runtime. This can quickly lead to an explosion in code complexity. Fortunately the Pilar representation already breaks every expression down to its smallest parts by making assignments to temporary variables. This means an assignment

```
x= 1 + 2 + 3
```

will translate to

```
t1 = 1
t2 = 2
t3 = 3
t4 = t1+t2
t5 = t4 + t3
x = t5
```

While the translated code takes more lines of code and performs more assignments on execution, it is much less complex to traverse in our various analyses. Another advantage is that it allows us to be more granular in our wrapping and unwrapping of variables. The increase in granularity leads to more efficient and more readable Simp code. We will now cover a number of translations from Guardol to Simp

8.1 Node Declaration

We start our translation with the node signature. The Guardol node signature

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

translates to

```
function MIME_Check(Input : in MIME_Type)
returns __gdlRet : [ t0 : MIME_Type_option; t1 : AuditMsg_option]   =
```

The translation looks very similar to the Guardol with a single exception; Simp doesn't allow multiple return values from its functions. To allow Guardol-style returns, we always return a record named `__gdlRet`. Simp doesn't have a representation of tuples, so we simulate them with a record that has field names `t0, t1, ..., tn`.

8.2 Local Declaration

Next we need to declare the locals. Many of the locals will look very similar to the locals in Guardol. As an example the Guardol locals are

```
virusOk : bool;
newAttachments : AttachmentList;
newBody : string;
attachAudit : AuditMsg;
dirtyAudit : AuditMsg;
```

and the first five Simp locals are

```
virusOk : MIME::bool;
newAttachments : MIME::AttachmentList_option;
newBody : MIME::string_option;
attachAudit : MIME::AuditMsg_option;
dirtyAudit : MIME::AuditMsg_option;
```

In the Simp translation, however, we declare the temp variables that we will use later in the node. Because we have already run our type checker, we know their types. The problem is that unnamed types can exist within expressions. The only place that these unnamed types are created is when a node is called and the node returns a tuple. An example is

```
newBody, dirtyAudit := DIRTY_WORD_SEARCH(Input.body) when virusOk;
```

The `DIRTY_WORD_SEARCH` node returns a tuple of type `(string * AuditMSG)`. The value of the return won't be bound to a name unless it is deconstructed and bound to separate variables or assigned to a variable that has a named type that it is compatible with. In Simp we will want to assign the result of the call expression directly to a temp variable. The difficult part is that temp variables aren't always associated with named types. Simp allows local variables to be assigned anonymous record types, so it isn't difficult for total

anonymous tuples. The problem is when they become partial, because it is impossible to create an anonymous union. To work around this we need to come up with a method for naming anonymous partial tuples. We do this by simply attaching the names of the tuple types and adding underscores between them. An example of such a type declaration is

```
type __gdl_anon_tup_gdl_primitives_string_option_AuditMsg_option_option = { SOME :
```

While the type name is a little long, it guarantees a unique type name for each anonymous tuple type. It is also very unlikely to have the same name as any types declared in the Guardol code.

8.3 Initilization

In the beginning of the node body we initialize every partial variable to its none value. We do this because Simp doesn't allow undefined variables in expressions but Guardol does. This only needs to be done for partial variables because our analysis has already shown that no total variable is ever referenced before it is assigned.

8.4 Node call

The first line of Guardol in the `MIME_Check` node is

```
virusOk := VIRUS_CHECK(Input);
```

This translates to

```
__temp2 := Input;
__temp1 := VIRUS_CHECK(MIME_Type_option'SOME(__temp2));
virusOk := __temp1;
```

in Simp. The only part is very different is the wrapping of `__temp2`. This needs to be done because `__temp2` is a total variable being passed into a node as a partial parameter. In Simp we need to wrap a variable whenever it is moved from total to partial.

8.5 When

The next line of Guardol code performs a `DIRTY_WORD_SEARCH` if the `Input` passed the virus check.

```
newBody, dirtyAudit := DIRTY_WORD_SEARCH(Input.body) when virusOk;
```

The translation is in Figure 8.2. The first two lines simply perform the dirty word search. After that there is an if statement. The if statement assigns the result of the dirty word search if the value of `__temp7` (which is the same as `virusOk`) it true. A problem with the translation is that the dirty word search is executed before we check the value of `virusOk`. This could lead to a situation where the dirty word search is run even though it isn't needed. Because there are no side effects in Guardol, this is only a problem of efficiency. The program that is generated is still correct.

```
__temp6 := Input;
__temp5 := __temp6.body;
__temp4 := DIRTY_WORD_SEARCH(__temp5);
__temp7 := virusOk;
/*when with total rhs*/
if __temp7 then
      __temp3:=__gdl_anon_tup_gdl_primitives_string_option_AuditMsg_option_optior
else
      __temp3:=__gdl_anon_tup_gdl_primitives_string_option_AuditMsg_option_optior
match __temp3 with
begin
      'SOME fresh0 => begin
                        newBody := gdl_primitives::string_option 'SOME(fresh0.t0);
                        dirtyAudit := AuditMsg_option 'SOME(fresh0.t1);
                  end
      'NONE => begin
                        newBody:= gdl_primitives::string_option 'NONE;
                        dirtyAudit:= AuditMsg_option 'NONE;
                  end
end
```

Figure 8.2: Simp translation of when

8.6 Tuple Deconstruction

We now have `__temp3` with the result of the when expression and we need to deconstruct it. We know `__temp3` is partial because it is the result of a when expression. This means we need to determine if it is defined or not before we perform the assignment. To do this we match `__temp3` with `SOME` or `NONE`. If we match `SOME` we put the value it contains into `fresh0`. `fresh0` is a variable the translation creates to hold values extracted by match statements. Once we have the unwrapped version of the record we can deconstruct it. We know that this record was a tuple in Guardol, so the naming scheme for the record's fields will be `t0`, `t1`, etc. We then go through the left hand side's names in order, assigning each a value from the tuple. If the tuple is undefined, we do the same thing only we assign the `NONE` values of the left hand side's respective types.

8.7 Type Constructions

The construction of a type is fairly simple if all of its components are total. We simply have to transform the Guardol syntax into the Simp syntax. When at least one variable in a type construction is partial however, we have to simulate Guardol's strict semantics in Simp.

```
Output := MIME_Type'[body => newBody, attachments => newAttachments];
```

is a record construction where both of the values going into the record are total. The translation is in Figure 8.3. The strict semantics of Guardol state that if any value in a

```
__temp12 := newBody;
__temp13 := newAttachments;
match __temp12 with
begin
      'SOME fresh1 => begin
                      match __temp13 with
                      begin
                              'SOME fresh2 => begin
                                              __temp11:=MIME_Type_option 'SOME([
                                      end
                              'NONE => begin
                                              __temp11:= MIME_Type_option 'NONE;
                                      end
                      end
                end
      'NONE => begin
                      __temp11:= MIME_Type_option 'NONE;
                end
end
Output := __temp11;
```

Figure 8.3: Simp translation of partial record construction

data-type construction is undefined, the entire datatype will be undefined. Figure 8.3 shows that we check to see if each value in the construction is defined. We only construct a new record if they are both defined. If either are undefined we assign the `NONE` value to the temp variable representing the construction expression (`__temp11`).

This does not cover the translation of every construct in Guardol, but it does introduce some of the most interesting aspects of the translation. The complete translation of the `MIME_CHECK` node is in figures 8.4 - 8.6

```
function MIME_Check(Input : in MIME_Type)
returns __gdlRet : [ t0 : MIME_Type_option; t1 : AuditMsg_option]   =
begin
 var
          virusOk : MIME::bool;
          newAttachments : MIME::AttachmentList_option;
          newBody : MIME::string_option;
          attachAudit : MIME::AuditMsg_option;
          dirtyAudit : MIME::AuditMsg_option;
          __temp1 : bool;
          __temp2 : MIME_Type;
          __temp3 : __gdl_anon_tup_gdl_primitives_string_option_AuditMsg_option_op
          __temp4 : [ t0 : string; t1 : AuditMsg];
          __temp5 : string;
          __temp6 : MIME_Type;
          __temp7 : bool;
          __temp8 : [ t0 : AuditMsg; t1 : AttachmentList];
          __temp9 : AttachmentList;
          __temp10 : MIME_Type;
          __temp11 : MIME_Type_option;
          __temp12 : gdl_primitives::string_option;
          __temp13 : AttachmentList_option;
          __temp14 : AuditMsg_option;
          __temp15 : AuditMsg_option;
          __temp16 : AuditMsg_option;
          __temp17 : AuditMsg_option;
          __temp18 : AuditMsg;
          __temp19 : bool;
          __temp20 : bool;
          __temp21 : gdl_primitives::string_option;
          __temp22 : AuditMsg_option;
          __temp23 : AuditMsg;
          __temp24 : bool;
          __temp25 : bool;
          Output : MIME_Type_option;
          Audit : AuditMsg_option;
in
  newAttachments := AttachmentList_option'NONE;
  newBody := gdl_primitives::string_option'NONE;
  attachAudit := AuditMsg_option'NONE;
  dirtyAudit := AuditMsg_option'NONE;
  __temp3 := __gdl_anon_tup_gdl_primitives_string_option_AuditMsg_option_option'NO
  __temp11 := MIME_Type_option'NONE;
  __temp12 := gdl_primitives::string_option'NONE;
  __temp13 := AttachmentList_option'NONE;
  __temp14 := AuditMsg_option'NONE;
  __temp15 := AuditMsg_option'NONE;
  __temp16 := AuditMsg_option'NONE;
  __temp17 := AuditMsg_option'NONE;
```

Figure 8.4: Simp translation of the `MIME_CHECK` node

```
__temp21 := gdl_primitives::string_option 'NONE;
__temp22 := AuditMsg_option 'NONE;
Output := MIME_Type_option 'NONE;
Audit := AuditMsg_option 'NONE;
__temp2 := Input;
__temp1 := VIRUS_CHECK(MIME_Type_option 'SOME(__temp2));
virusOk := __temp1;
__temp6 := Input;
__temp5 := __temp6.body;
__temp4 := DIRTY_WORD_SEARCH(__temp5);
__temp7 := virusOk;
/*when with total rhs*/
if __temp7 then
      __temp3:=__gdl_anon_tup_gdl_primitives_string_option_AuditMsg_option_optior
else
      __temp3:=__gdl_anon_tup_gdl_primitives_string_option_AuditMsg_option_optior
match __temp3 with
begin
      'SOME fresh0 => begin
                        newBody := gdl_primitives::string_option 'SOME(fresh0.t0);
                        dirtyAudit := AuditMsg_option 'SOME(fresh0.t1);
                  end
      'NONE => begin
                        newBody:= gdl_primitives::string_option 'NONE;
                        dirtyAudit:= AuditMsg_option 'NONE;
                  end
end
__temp10 := Input;
__temp9 := __temp10.attachments;
__temp8 := attachmentListCheck(__temp9);
attachAudit := __temp8.t0;
newAttachments := __temp8.t1;
__temp12 := newBody;
__temp13 := newAttachments;
match __temp12 with
begin
      'SOME fresh1 => begin
                        match __temp13 with
                        begin
                              'SOME fresh2 => begin
                                                __temp11:=MIME_Type_option 'SOME([
                                              end
                              'NONE => begin
                                                __temp11:= MIME_Type_option 'NONE;
                                              end
                        end
                  end
      'NONE => begin
                        __temp11:= MIME_Type_option 'NONE;
```

Figure 8.5: Simp translation of the `MIME_CHECK` node continued

```
                end
    end
    Output := __temp11;
    __temp15 := attachAudit;
    __temp18 := AuditMsg'Dirty_Word_Check_Failed;
    __temp21 := newBody;
    /*Exists*/
    match __temp21 with
    begin
        'SOME fresh3 => begin
                __temp20 := true;
           end
        'NONE => begin
                __temp20 := false;
           end
    end
    __temp19 := not __temp20;
    /*when with total rhs*/
    if __temp19 then
          __temp17:=AuditMsg_option'SOME(__temp18);
    else
          __temp17:=AuditMsg_option'NONE;
    __temp23 := AuditMsg'Virus_Check_Failed;
    __temp25 := virusOk;
    __temp24 := not __temp25;
    /*when with total rhs*/
    if __temp24 then
          __temp22:=AuditMsg_option'SOME(__temp23);
    else
          __temp22:=AuditMsg_option'NONE;
    match __temp17 with
    begin
          'SOME fresh4 => begin
                          __temp16 := __temp17;
                end
          'NONE => begin
                          __temp16 := __temp22;
                end
    end
    match __temp15 with
    begin
          'SOME fresh5 => begin
                          __temp14 := __temp15;
                end
          'NONE => begin
                          __temp14 := __temp16;
                end
    end
    Audit := __temp14;
    __gdlRet := [ t0 : Output,t1 : Audit ];
end
```

Figure 8.6: Simp translation of the `MIME_CHECK` node continued

Chapter 9

Conclusion

The Guardol compiler is currently in a solid state. We have a solid collection of test cases separated to test each part of the language as designed so far. The test runs the entire tool-chain and reports any unexpected results. Half of the tests are pass cases, which are expected to terminate without producing any errors. The remainder of the tests are fail cases which should generate errors (but not crash). The compiler currently passes all of these tests.

There are still parts of the compiler that could use improvement, many because the language design is not fully mature. One open question in the Guardol project is what type of recursion to allow. We certainly need to allow a method to call itself for the purpose of deconstructing list data-types. We are uncertain, however, if we wish to allow for more complex recursion. Limiting the recursion capabilities of the language can give significant advantages to the verification and analysis of the program, but it can also limit the number of behaviors that Guardol is capable of defining.

Another issue that is part language design is the use of multiple files to define Guardol behavior. Currently a single Guardol file can have multiple packages in it. It is very useful for a developer to be able to locate different packages in different files, and even organize those by folder. Guardol currently lacks a construct to declare file dependencies. Once the construct is created the compiler would need to be modified up to the Pilar translation. Most of the work will be in the symbol table because it needs to be able to resolve every name used in the Guardol code.

There are a number of other features that would make Guardol applications much easier to develop. Guardol is meant to be translated into a variety of languages and run on a variety of platforms. This means that Guardol needs an external environment for code to be executed. Work has been started on a Guardol simulator, but it runs from the AST and can't take full advantage of our static analyses. A Guardol simulator would save a developer time, and allow simulations to run in a faster (and sometimes more transparent) environment. The next step from a successful simulator would be a debugging environment, which would also be incredibly useful to Guardol developers.

Because Guardol is going to be translated into a number of languages for use on various platforms, it is important to make the current GUI and the compiler extensible. The

compiler currently has extensibility built in in the form of external types, constants, and nodes, but there is currently no easy way to develop unique plug-ins for new languages and attach them to the current plug-ins. These plug-ins could allow for translations and test runs, or to run any analysis that might already come with the language Guardol is being translated to. Another point that is important is to create a framework that allows developers to easily write new translations from Guardol to their target language. A large part of the challenge here will be making the results of our analyses more readily available to an external developer.

There are more interesting things that could be done with the GUI, some of which are features already available in XText. XText has the capability to build new project wizards and code templates. Both of these features can be very helpful both to developers that are new to the language, and those that are skilled enough to leverage their full power.

Bibliography

[1] Unified Cross Domain Management Office, What is a cross domain solution?, http://www.ucdmo.gov/faqs.html.

[2] R. E. Pollock, Radiant mercury simulation capabilities, Technical report, Lockheed Martin.

[3] General Dynamics, Tactical cross domain solution (TCDS), 2010.

[4] Owl Computing Technology Inc., ECDS, 2010.

[5] Tenix America, Tenix america cross-domain solution receives full accreditation, Business Wire : http://www.businesswire.com/portal/site/home/permalink/?ndmViewId=news_view&newsId=20070910005385&newsLang=en, 2007.

[6] Boeing, Boeing receives government security certification for secure network server, http://www.boeing.com/news/releases/2007/q2/070621b_nr.html, 2007.

[7] Xtext, Xtext, http://www.eclipse.org/Xtext/, 2010.