UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

AUTOMATIC CONVERSION OF ADA SOURCE CODE TO SCALA

Guilherme Jorge Nunes Monteiro Espada

Mestrado em Engenharia Informática
Especialização em Arquitetura, Sistemas e Redes de Computadores

Dissertação orientada por:
Prof. Doutor Alcides Miguel Cachulo Aguiar Fonseca

2020

Acknowledgments

Nunca fui uma pessoa de muitas palavras, e esta secção não é exceção: quero agradecer à minha mãe, por ter dado tudo para eu ter esta oportunidade. Quero agradecer ao meu orientador, Alcides, por me ter puxado ao longo do ano para superar os meus limites. Quero agradecer às *actividades de integração académica não organizadas pela faculdade* por me darem colegas. E finalmente quero agradecer ao *gang* dos almoços, em especial ao Medeiros e ao Paulo, por me darem motivação para vir todos os dias para a faculdade.

This work was supported by FCT through the LASIGE Research Unit, ref. UIDB/00408/2020.

Dedicado às minhas Avós

Resumo

Ada foi criada nos anos 70 pelo Departamento de Defesa dos Estados Unidos como uma linguagem que pudesse ser utilizada para todos os projetos do Departamento de Defesa. Durante os anos 80 ganhou popularidade devido à sua versatilidade. No entanto, pelos anos 90, a sua popularidade começou a diminuir. Isto levou à situação atual: existe código Ada ainda em operação, mas não existem os recursos humanos necessários para suportar tal código. Adicionalmente, mesmo que existam os recursos para suportar o código, desenvolver novas funcionalidades torna-se mais dispendioso quão menos popular a linguagem for. Para além disso, pode ser desejável para o programador utilizar certas funcionalidades não disponíveis em Ada, mas disponíveis numa linguagem mais moderna.

Em todos estes casos, uma das soluções possíveis passa por converter o código Ada para uma linguagem mais moderna. Uma possíveis métodologias para a tradução de código para outra linguagem é a chamada *abstração-reimplementação*. Segundo este método, produz se uma descrição de alto nivel do funcionamento do código original, e seguidamente reimplementa-se o código a partir da descrição. Este processo, apesar de produzir código bastante legível e canônico, tambêm é bastante propenso a erros, visto que grande parte da intenção do código original é perdida no processo.

Em alternativa existe outro modelo de conversão: o *modelo dos halteres*. Neste modelo, a transformação tem três fases: normalização, tradução, e otimização, sendo a tradução entre duas linguagens realizada linha a linha na fase de tradução. A fase de tradução é a mais débil, visto que não é possível testar um projeto meio convertido. No entanto, manualmente reescrever o código linha a linha pode ser bastante dispendioso, ambos em termos monetários e em termos temporais. Assim sendo, torna-se natural tentar automatizar este passo do processo.

Compilação fonte-a-fonte, também conhecida como transpilação, descreve uma técnica em que o compilador, em vez de ter como alvo linguagens máquina do tipo *assembly* ou *bytecode*, tem como alvo código fonte (geralmente numa linguagem diferente da original). Transpilação, por exemplo, viu uso como ferramenta para portar programas para arquiteturas mais recentes, ou bibliotecas para ambientes não suportados. No entanto, estas ferramentas são inapropriadas para o nosso uso, visto que muitas vezes negligenciam a legibilidade do código.

Ada em particular inclui certas funcionalidades que não podem ser traduzidas trivialmente visto que não existem funcionalidades diretamente equivalentes em linguagens mais modernas. Por exemplo, nenhuma linguagem que esteja no top 10 do índice TIOBE em 2020 suporta tipos de virgula fixa, ou delimitados nativamente. Similarmente, nenhuma linguagem (à exceção de Ada) que esteja no índice suporta o sistema de tarefas como existe no Ada. No entanto, estas funcionalidades são frequentemente usadas em código Ada. Visto isto, é essencial que estas funcionalidades sejam traduzidas. Porém, traduzir estas funcionalidades imprudentemente pode causar uma explosão em tamanho de código e complexidade. É claro, então, que estas funcionalidades têm de ser traduzidas cuidadosamente de forma a minimizar a complexidade e maximizar a manutenibilidade.

Trabalho prévio nesta área tende a dividir em dois campos: Ou na correção semântica, como é o caso quando a transpilação é usada como um passo de compilação para tomar proveito do alcance de outro compilador. Ou na legibilidade do código, como quando é utilizada no trabalho prévio em tradução mecanizada de código. Trabalho prévio foge de um para o outro, em vez de tentar atacar ambos ao mesmo tempo. Neste trabalho, através de seleção cuidadosa da linguagem alvo, e uma consideração minuciosa de como as características interagem entre si, conseguimos manter ambos legibilidade e correção semântica.

Este trabalho começa por apresentar um estudo de possíveis linguagens alvo para transpilação a partir de Scala. Ao escolher uma linguagem alvo, é importante ter em consideração a extensão da sobreposição entre as funcionalidades das duas linguagens. Nomeadamente, o subconjunto de características que ambas as línguas suportam nativamente da mesma forma são as únicas características que vão ser perfeitamente suportadas. No entanto, este conjunto é muitas vezes mínimo. Como tal, também é preciso ter em mente como as características da linguagem alvo vão ser emuladas na língua de fonte. As características da linguagem de destino podem afectar fortemente a forma como as características da linguagem origem podem ser emuladas, e consequentemente, a legibilidade do código convertido. Deste processo destaca-se Scala como linguagem alvo devido ás suas características que auxiliam a implementação de linguagens especificas de domínio. As características mais importantes são o suporte para macros, que permite reduzir drasticamente a quantidade de *boilerplate* que é mostrado ao programador; as conversões implícitas personalizadas, que permitem modificar a maneira como os tipos interagem uns com os outros; a definição de operadores personalizados, que permite que o programador não tenha de invocar métodos verbosos para até operações simples como a soma e finalmente a segurança de memoria e a verificação de tipos em tempo de compilação, que permite ao programador ter segurança sobre o funcionamento do seu código.

De seguida apresento as regras para a conversão de várias características para Scala. Primeiramente os tipos delimitados e de vírgula fixa, comummente usados para qualquer operação envolvendo números em Ada. De seguida os *records*, o método mais comum de agrupar variáveis em Ada. O sistema de tarefas, que é o método de eleição para realizar *multithreading*. Seguindo para os parâmetros com modificadores de direção, característica única da Ada. Passando pelas matrizes, a base de várias estruturas de dados. E finalmente apresentando as regras para genéricos, característica que promove a reutilização do código. Apresenta também um protótipo que implementa estas regras.

Posteriormente, usando apenas as características abordadas, apresentamos vários programas de teste, que testam as várias características da conversão. Estes programas de teste são também utilizados como parte de um inquérito que pretende descobrir a qualidade das traduções seguindo as regras previamente apresentadas. Com este inquérito conseguimos concluir que os programas resultantes da conversão são ambos legiveis, e similares aos programas em Ada. Esta similaridade é essencial para manter a familiaridade dos programadores com o código.

No entanto, ainda existe mais trabalho futuro em aumentar a cobertura da linguagem Ada. O protótipo implementado não suporta grande parte da biblioteca padrão do Ada e ainda existem muitas características que podem ser adicionadas, como por exemplo: os primitivos de sincronização, úteis para programas *multithreaded* e pre- e pos- condições em métodos, úteis para a analise estatica de programas. As declarações *delay* e *timed*, úteis para programas que têm de correr em tempo real.

Finalmente, embora este trabalho se concentre na tradução especificamente entre Ada e Scala, há conclusões gerais que podem ser tiradas deste trabalho. Em particular, Scala é uma língua alvo adequada para a tradução de código que preserva a legibilidade. As funções implícitas permitem que o código traduzido esconda transformações de que se revelariam onerosas para o programador. O suporte de macros permite a geração de código durante a compilação, reduzindo assim a quantidade de *boilerplate* mostradas ao programador. Finalmente, a definição de operadores personalizados, permite omitir chamadas de funções verbosas melhorando ainda mais a legibilidade do código.

Palavras-chave: Compiladores, Linguagens de Programação, Concurrencia, Engenharia de Software, Confiabilidade de Software

Abstract

When the popularity of a programming language declines, there are often sizeable existing legacy codebases that need to be modernized.

However, migrating off of these legacy languages is very costly. In addition, these declining languages do not necessarily have the same feature set as a modern language. As such, there are often specific features only present in the legacy language, which are useful for a variety of applications, making it hard for modern languages to penetrate these domains.

This thesis presents a study of how these advanced features in legacy programming languages can be implemented in modern languages. As a use case, we will consider Ada, a language which is declining in popularity, and we evaluate which modern languages can adequately express Ada's features. We select Scala as the most suitable language, due to its advanced type system, macro capabilities, implicit functions and constructors, and operator overloading.

In addition, we present a set of translation rules from Ada to Scala, which was implemented in an automatic translation tool.

Previous work on this domain tends to focus either on semantic correctness or on readability, eschewing one for the other, rather than tackling both at the same time. However, by carefully picking a target language with enough features that lend themselves for this kind of translation, we achieve a result that preserves both semantics and similarity, while maintaining an acceptable level of readability.

We expect that this result to reduce the time and monetary cost of migrating off these legacy platforms, thereby making automated conversion of large software systems more viable.

Keywords: Compilers, Programming Languages, Concurrency, Software Engineering, Software Safety

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

This work deals with the problem of automated conversion of source code between two programming languages. Section 1.1 explains the motivation and importance of the work. Chapter 2 introduces the Ada-specific difficulties in source translation. Section 1.3 introduces the challenges in automated translation, as a whole. Section 1.4 discusses how these general issues apply to Ada. Section 1.5 and section 1.6 respectively define the goals and contributions of the work. Finally, section 1.7 presents the structure of this document.

1.1 Motivation

Ada was created in the '70s by the United States' Department of Defense as a programming language that could be used for every department project [24]. Ada intially targeted the development of embedded systems, but saw broader use, primarily due to it becoming mandatory in new Department of Defense projects.

Ada was popular in the '80s as a general-purpose language. However, by the late '90s, its popularity started to decline (fig. 1.1). This has led to a situation where there is legacy code still in operation, but there is not enough human resources to support it [24]. Besides, even if an organization has the manpower to continue the development of their project in Ada, the cost of developing new features might be too high due to the stagnating ecosystem. Furthermore, one might need features that are not available as a library in Ada but exist in other more modern languages.

In all these cases, moving away from Ada towards a more popular language might be beneficial. However, manually rewriting the existing codebase can take an impracticable amount of time and effort [26]. In this scenario, source-to-source compilation provides an efficient solution: by automating the process, the amount of time and effort spent can be drastically reduced.

Source-to-source compilation (also known as transpilation) is a technique wherein a compiler, instead of targeting assembly or bytecode languages, targets another source language. Transpilation has, for example, seen uses as a tool to port older programs to

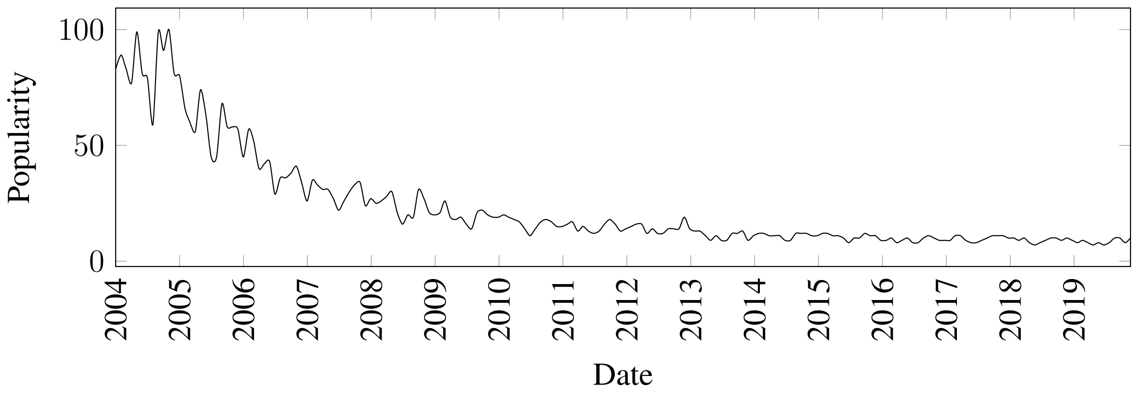

Figure 1.1: Popularity of Ada over time, according to Google Trends.

newer architectures [14] and libraries to unsupported environments [28]. However, our efforts are quite different from these, since we want to produce human-readable code.

Ada includes some features that cannot be trivially translated due to suitable alternatives not being present in more modern language. However, these features are often present in legacy code, so it is essential that they are translated. Carelessly emulating these features can cause an explosion in code size and complexity. These features must then be carefully implemented in order to minimise complexity and maximise maintainability.

This work focuses on implementing these features in a way that is both readable and semantically compatible with "legacy" languages.

In addition, this work was motivated by and attracted interest from a large international bank.

1.2 Software Migration Methodologies

There is a general technique for software migrations in the literature, often dubbed "Abstraction-Reimplementation" [30].

This technique consists of converting a program by first analysing the source code to produce a high level description of the behaviour and then reimplementing the program from the high level description (as shown in fig. 1.2). While this process tends to produce higher quality, more native looking code than the alternative model, many bugs can be re-introduced as the difference between an implementation detail and an edge case is often hard to discern. Furthermore, it is not generally possible to test until the very end of the process: the specification cannot be tested, and partial implementations can only be partially tested.

Due to this, both producing a high level description and concretizing the translated code from the description are error-prone, subjective tasks.

Instead of this model, we opt for the Barbell Model, as shown in fig. 1.3. The Barbell Model [21] identifies three phases of software conversion. Firstly, the normalization

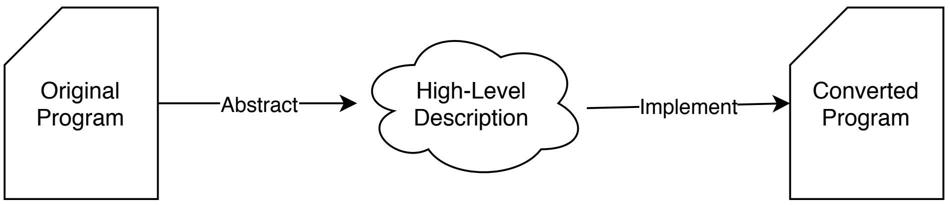

Figure 1.2: The Abstraction-Reimplementation migration methodology.

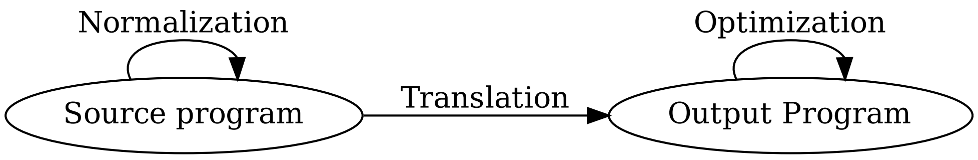

Figure 1.3: The barbell migration methodology.

consists of preparing the software for conversion, and simplify the next step. For example, some features in particular may be difficult to translate. In this case, refactoring the code not to use that feature would be beneficial.

Secondly, translation consists of actually transforming the code from the source language from the target language line by line. This is the most challenging and most fragile part: while the ends of the barbell can be tested, it is much more difficult to test a half-converted piece of software.

Finally, the code that results from translation is likely not in the native style of target language. In addition, to aid in translation, and as a result of the first step, layers of indirection are often employed. Optimization consists of the final clean-up of the code in the target language.

This work tackles the most fragile part of the process: the translation.

1.3 The Realities of Language Conversion

According to "The Realities of Language Conversion" [26], language conversion has a tantalizingly simple problem statement: "Convert this system to that language without changing the external behaviour."[26]. Furthermore, this seems to be a relatively simple problem to solve: Converting `Result := Result * 2;` to the equivalent Scala code `result = result * 2` is trivially. And indeed, that specific instance, like many similar ones, is easy to solve. However, the devil is in the details. The availability of constructions that facilitate the expression of a solution determines the type of problem that is easy to formulate a solution to. For instance, if one wishes to express switch-like[1] behaviour, then a language that has a switch statement is more convenient. If, on the other hand, the language does not have a switch statement, then one has to be simulated using other native constructs (for example, an if-else chain).

The language conversion problem consists of mapping the input language to native and simulated constructs in the output language [26]. Ideally, only native constructs in the output language would be used. However, converting simulated constructs to native constructs is undecidable in the general case, and might require heavily restructuring of the code. Instead, a more achievable goal is to convert while minimizing the amount of new simulated constructs introduced. Besides converting to native or simulated constructs, there is also the option of not supporting the construct. This may be the correct choice, for example, if supporting a construct would imply making the support of another feature less canonical.

Another issue of language conversion is that the code produced from conversion does not instantly look like code from the target language ecosystem. Instead, it looks like code from the source language ecosystem, in a different language. This is an issue that is difficult to solve automatically, since automated restructuring is a challenging problem, and what "looks like code from the ecosystem" is often not subject to hard rules, and instead acquired over years of experience with the language.

1.4 Simulation of language constructs

If the source and target language are similar enough, then translation is trivial: swap the source language syntax with the target language syntax. However, most source-target language combinations do not enjoy this property. In that case, there are two possible outcomes for each unsupported construct: either don't support it (supporting it with incorrect behaviour is virtually the same as not supporting it since programs that depend on such behaviour will run incorrectly in the new environment) or simulated it.

A excellent example of this is Ada's fixed-point types. Most modern languages do not support fixed-point types. As such, these types can either be converted to native floating-point types (possibly introducing rounding errors) or be converted to a new, type that simulates Ada's behaviour. Of note is that depending on the chose target language, emulating these types might diminish the readability of the generated code. If, for instance, the target language does not support operator overloading, then the native arithmetic operators cannot be used with our custom types, heavily diminishing readability.

[1]Switch-like behavior means behaviour similar to choosing between many code paths based on a single value.

1.5 Goals

The goal of this work is to design and evaluate a methodology for automatically translating source code in a legacy, yet feature-rich, programming language to a more modern one.

We intend to implement a prototype tool that is able to perform such conversion, following the following desired properties:

- Accepts a subset of the Ada language.
- The translation preserves the safety properties of Ada.
- The resulting code should maintain similarity with the original code.
- The resulting code should be as readable as possible, without compromising the previous goals.

Alongside the tool, it is also a goal to implement related runtime libraries necessary for the converted code. It is explicitly outside the scope of this work to accept the full Ada language, as that effort is better spent focusing on the more "interesting translations". It is also not a goal to produce code completely canon to the target language: instead, we opt to produce code similar to the input, in order to maximize knowledge transfer.

1.6 Contributions

This work gives the following contributions:

- Determine a suitable language to use as a compilation target for Ada code.
- Select (and determine the encoding for) a subset of features which are both commonly used and can be reasonably encoded in the target language (Scala), namely:
  - Ranged and fixed-point types
  - Records
  - Tasking system
  - In and Out parameters
  - Arrays
  - Generics
- Implementation of a tool that automatically converts Ada code to Scala based on the encodings defined in the previous item.
- Evaluation of the quality of the generated code.

1.7 Structure of the document

This document is organised as follows:

- Chapter 2 - Ada's Peculiarities
- Chapter 3 - Related Work
- Chapter 4 - Evaluating Candidate Target Languages
- Chapter 5 - Ada to Scala translation
- Chapter 6 - Evaluation
- Chapter 7 - Conclusion & Future Work

Chapter 2

Ada's Peculiarities

Ada is a strongly statically typed language like Java, C#, Scala (and other modern languages). However, Ada differs from most other languages in some areas, which make it challenging to translate to modern languages.

2.1 Numerics

Ada requires explicit conversions between types that are usually automatically handled by modern languages. For example, the implicit integer to decimal conversion is explicit in Ada, as exemplified in listing 2.1.

```
float n = 1 + 1.0; // C
```

```
float n = 1 + 1.0; // Java
```

```
n = 1 + 1.0 # Python
```

```
N: Float := Float(1) + 1.0; -- Ada, Explicit!
```

Listing 2.1: Conversion from a integer to a double. Only in Ada is the conversion explicit.

Another feature that is uncommon in modern languages is fixed-point number support. Ada supports fixed-point numbers with any precision or range, specified by the programmer. Ada even allows programmers to specify precisions that are not powers of two (a feature that is uncommon even among fixed-point libraries). Programs often use these fixed-point types because they frequently can have more precision than a conventional floating-point number. See listing 2.2 for an example of this behaviour.

In addition, Ada allows the programmer to easily define their own types which can only hold a specific subset of values of another type. We refer to these as "ranged types". Variables of these types get their values checked for range every time they are assigned. Ada also permits implicit conversions between these subtypes (but not between subtypes

```
>>> from FixedPoint import FXnum
>>> # Correct result is 3.142857142857142857142857142...
>>> 22/7 # 64bit floating point
3.142857142857143
>>> print(FXnum(22) / FXnum(7)) # 64bit fixed point
3.142857142857142857127
```

Listing 2.2: In Python: floating-point operation with less precision than the same-sized fixed-point operation.

of different types, as shown in listing 2.3).

```
type SmallInt is range 1 .. 10;
A : SmallInt := 2;
B : SmallInt := 20; -- Error!
type OtherSmallInt is range 1 .. 10;
C : OtherSmallInt := A; -- Error!
D : OtherSmallInt := OtherSmallInt(A); -- Explicit cast, OK!
subtype SmallerInt is SmallInt range 1 .. 3;
E : SmallerInt := A; -- Implicit cast between subtypes, OK!
E : SmallerInt := 8; -- Error!

type Money is delta 0.01 range 0.0 .. 1000.0; -- Fixed point
```

Listing 2.3: Demonstration of ranged types in Ada.

2.2 Tasks

Ada has a unique Task system for multiprocessing, not found in any of the surveyed modern languages found in chapter 4. Since the Task System is the *de facto* way to do parallelization and concurrency in Ada, its simulation is essential to converting real-world programs.

Each task represents a thread of execution that proceeds independently and concurrently among the sections where it interacts with other tasks. A task is started (and initialized) when it is created. By default, Ada code executes in the main task. All tasks, except the main task, have a parent, which corresponds to the task that created them. A parent task cannot terminate until all child tasks either terminate or are in a state where they are terminable (section 2.2.2). A simple task is shown in listing 2.4.

2.2.1 Inter-Task Communication

A task by itself, without communication with other tasks, has limited uses. To solve this, Ada allows tasks to specify points in which they can synchronise with another task and

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Task_Print_Demo is
   task PrintTask;
   task body PrintTask is
   begin
      Put_Line ("Printing from print task");
   end PrintTask;
begin
   Put_Line ("Printing from main task");
end Task_Print_Demo;
```

Listing 2.4: Simple Ada task. The order in which the strings are printed is undefined.

receive or transmit data. These are referred to as "entry" points.

An entry point is defined in a task's header. Then, another task can call this entry point. The calling task is blocked until the task accepts the entry. If the task accepts an entry that hasn't been called yet, then the task blocks until the entry is called. After the accept block, both tasks (caller and callee) resume asynchronous execution. Listing 2.5 shows an example of a task with an entry. Both tasks print "1", asynchronously. Then, the tasks *rendez-vouz* and "2" is printed by the callee task, while the caller is blocked. Afterwards, both tasks asynchronously print "3".

```
with Ada.Text_IO; use Ada.Text_IO;
procedure S is
   task PrintTask is
      entry Sync;
   end PrintTask;
   task body PrintTask is
   begin
      Put ("1");
      accept Sync do Put ("2"); end;
      Put ("3");
   end PrintTask;
begin
   Put ("1");
   PrintTask.Sync;
   Put ("3");
end S
```

Listing 2.5: Ada task with entry. "11233" is always printed.

Additionally, actions can have parameters. The behaviour of these parameters is similar to method parameters, as explained in section 2.3.

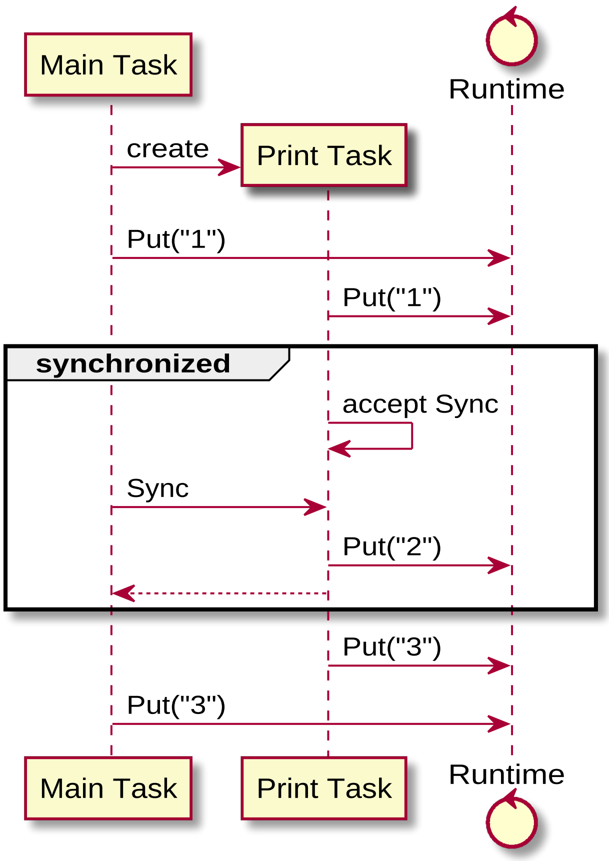

Figure 2.1: Diagram of the execution of the program in listing 2.5.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure P is
   task T is
      entry Print( i : in out Integer );
   end T;
   task body T is
   begin
      accept Print( i : in out Integer ) do
         i := i + 1;
         Put (i);
         i := i + 1;
      end;
   end T;
   AnInt : Integer := 1;
begin
   Put (AnInt);
   T.Print(AnInt);
   Put (AnInt);
end P;
```

Listing 2.6: Ada task with in out parameter. "123" is always printed.

2.2.2 Accepting between different alternatives

It is sometimes useful for a program to accept many possible entries, depending on what actually gets invoked. For this purpose, the Ada language has the select statement. The select statement will block until it is able to fulfil one of its accept alternatives.

Furthermore, it is useful to specify a condition that must be met before a message can be accepted. The select statement also allows the programmer to specify a condition that must be met before a specific alternative is selected.

Likewise, it is useful to allow a child to terminate when its parent wishes to terminate. To achieve this, Ada allows `terminate` to be specified as one of the branches of the select. This indicates that when the task is blocked in such a select statement, it may break out of the select if the parent wants to terminate.

An example using these 3 features is shown in listing 2.7.

2.3 Parameters

Similar to other systems languages like C, Ada allows parameters to be passed by reference or by value. However, the behaviour is different than C.

Any parameter can be annotated with `IN`, `OUT` or `IN OUT`. By default, parameters are treated as if they have the `IN` specifier. These specify the direction of data flow.

```ada
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure P is
   task T is
      entry Print;
      entry Increment (Value : in Integer);
   end T;
   task body T is
      CurrValue : Integer := 0;
   begin
      loop
         select when CurrValue mod 2 = 0 =>
            accept Print;
            Put (CurrValue);
            Put_Line (" is even");
         or when CurrValue mod 2 /= 0 =>
            accept Print;
            Put (CurrValue);
            Put_Line (" is odd");
         or
            accept Increment (Value : in Integer) do
               CurrValue := CurrValue + Value;
            end Increment;
         or
            terminate;
         end select;
      end loop;
   end T;
begin
   T.Print;
   T.Increment (3);
   T.Print;
   T.Increment (7);
   T.Print;
end P;
```

Listing 2.7: Complex ada task.

`IN` parameters behave as any other pass-by-value parameter: the value gets copied into the function when the function is called. `OUT` parameters behave in reverse: the value gets copied out of the function when the function ends. The behaviour of `IN OUT` parameters is then intuitive: it combines both behaviours, copying in, and then copying out. An example is given in listing 2.8.

```ada
with Ada.Integer_Text_IO;

procedure InOutTest is
procedure Add (A: in out Integer; B: in Integer) is
begin
   A := A + B;
end Add;
    A: Integer := 2;
begin
    Ada.Integer_Text_IO.Put(A);
    Add(A, 3);
    Ada.Integer_Text_IO.Put(A);
end InOutTest;
```

Listing 2.8: Demonstration of in and out parameters.

2.4 Arrays

Arrays are data structures that can store a finite amount of elements of a specific type.

Ada's arrays have a distinction from other major languages: it allows arrays to start at any index desired by the programmer. In addition, Ada arrays support initialization (listing 2.9), slicing and copies (listing 2.10) with a terse syntax.

```ada
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure A is
   type R is range -1 .. 1;
   Arr : array (R) of Integer := (0 => 0, others => 1);
begin
   Arr := (0 => 1, others => 0);
   Put (Arr (-1));
   Put (Arr (0));
   Put (Arr (1));
end A;
-- Prints: 0, 1, 0
```

Listing 2.9: Array initialization syntax.

```ada
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure A is
   type R is range -1 .. 1;
   Arr : array (R) of Integer;
begin
   Arr (0)       := 1;
   Arr (1)       := 2;
   Arr (-1 .. 0) := Arr (0 .. 1);
   Put (Arr (-1));
   Put (Arr (0));
   Put (Arr (1));
end A;
-- Prints: 1, 2, 2
```

Listing 2.10: Array example.

2.5 Generic Programming

Generic programming refers to a style of programming wherein blocks of code can contain unknown types, but later instantiated as parameters.

Ada supports generic programming by allowing either functions or whole packages to be parameterized at compile time. This can be used, for example, to create a data structure that works for many element types.

Chapter 3

Related work

In this chapter we will present prior work related to software migration and automated translation.

3.1 The legacy systems migration dillema

The term “Legacy systems” can be defined as systems that are still running in day-to-day operations, but whose development has stalled [7]. This stall is usually due to, the age of the underlying language, system, or libraries. As the system ages, it becomes more and more difficult to find developers to work work on it (which also makes developers more expensive). Concurrently, it also becomes harder and harder to efficiently develop feature for it (which in turn, makes such features more expensive).

However, systems migration is not instant and effortless. As such, it can be expensive to migrate a large, running system. This is specially true for systems that use old languages, architectures, and libraries that cannot cleanly interface with a new system.

Thus a dillema arises: On one hand, the system is getting increasingly fossilized, and new features are harder to develop. On the other hand, the system does function, and migrating away can be a huge leap of faith that the new system will fulfill all the requirements of the old system [7].

3.2 Manual and automated translation

Traditionally, software translations are done manually. Any manual task is error-prone, and software translation is no exception. As such, it is natural to attempt to automate this task. In addition, automation can speed up and reduce the cost of translation.

Automated source to source translation has been much researched in order to overcome the limitations of manual translation. This has long been a standing topic in software engineering (see [14] for an Intel 8080 to Intel 8086 assembly converter from 1981), with much work being done with varying combinations of source languages, target languages and overarching goals and requirements.

Source to source translation is useful for a variety of applications, which can roughly be classified into three categories: as a migration tool, as a compilation step and as an analysis aid.

3.3 Translation as a migration tool

In [29], the authors convert Assembly to C via the Wide Spectrum Language language (an external DSL for program transformation work, which includes low-level programming constructs and high-level abstract constructs within a single language, as shown in listing 3.1). Due to the highly unbounded nature of Assembly behaviour, the authors had to make several assumptions about the behaviour of the input source. Several handwritten restructure operations are also applied to make the output code more readable. These operations are general, not program-specific. However, the output code still resembles unstructured assembly.

```
!P mvc(a[db(WRITEM; r3); 3 + 1]
↪  var a[db(WLAST; r3); 3 +
↪  1]);
```

(a) Low level sample.

```
WLAST := WREC.WRITEM;
```

(b) High level sample.

Listing 3.1: Wide Spectrum language examples. Both samples have the same effect.

Under this framework, the speed of translation is mostly irrelevant, while on the other hand, generated code quality is essential.

3.4 Translation as a compilation step

In [12], the authors use conversion from Smalltalk to Objective-C as a way to work around “Smalltalk’s deficiencies in the delivery of production-quality versions”. Similarly, in [32], the authors mention “By translating the Smalltalk code into portable and interoperable C code, it is possible to deliver a standalone version of Smalltalk applications and develop an application partly in Smalltalk, and partly in C”. However, handwritten application specific rules are used, which decreases the usability of the tool.

In [15], the authors use conversion as a way to “make a large body of well-tested Fortran source code available to C environments”. A complete tool is presented. However, the produced C code is not always portable, due to differences between the basic data types in C and in Fortran.

In [28], the authors present their work as “a solution for the reuse of Java code within Eiffel programs based on a source-to-source translation from Java to Eiffel”. They successfully convert libraries of varying sizes and call them from native Eiffel code.

X10 [10] is a highly parallel programming that compiles to C or Java. X10 has many features that do not have a direct, native mapping in either language[25]. As such, when using certain features, the code explodes in size and unreadability.

```
class C(p:Int) {
  val q:Int; var r:Int;
  def u() { }
  private def v() { }
  def w() { return super.hashCode(); }
  def this(p:Int) {
    property(p);
    q = 1;
  }
  static val s = Place.MAX_PLACES;
  static def t() { }
}
```

(a) X10 sample.

```
import x10.lang.Place;
import x10.lang.Runtime;
import x10.runtime.impl.java.InitDispatcher;
public class C extends x10.core.Ref {
  public int p, q, r;
  final public int p() { return this.p; }
  public void u() { }
  private void v() { }
  public static void v$P(final C C) { C.v(); }
  public int w() { return super.hashCode(); }
  final public int C$hashCode$S() { return
↪  super.hashCode(); }
  public C(final int p) {
    super();
    this.p = p;
    this.__fieldInitializers173();
    this.q = 1;
  }
  final private void __fieldInitializers173() { this.r = 0;
↪  }
  final public static void __fieldInitializers173$P(final C
↪  C) {
    C.__fieldInitializers173();
  }
  public static int s = 0;
  public static void t() { }
  public static int fieldId$s;
  public static int getInitialized$s() {
    if (Runtime.hereInt() == 0) {
      C.s = Place.getInitialized$MAX_PLACES();
      InitDispatcher.broadcastStaticField(C.s,
      ↪  C.fieldId$s);
    }
    return C.s;
  }
  public static void getDeserialized$s(byte[] buf) {
    C.s = (Integer) InitDispatcher.deserializeField(buf);
  }
  static {
    C.fieldId$s = InitDispatcher.addInitializer("C", "s");
  }
}
```

(b) Compiled to Java.

Listing 3.2: Comparison of source X10 code and output after compilation into Java.

Another example is CHICKEN Scheme [31], which compiles Scheme via C, via continuations.

In all these cases, the readability of the generated code is not important, but the speed of generation and generated code is, as the tool would be used as a build step [26]. An example of a translation that follows these principles is shown in listing 3.2. Another translation, this time of scheme, by the CHICKEN compiler is shown in fig. 3.1.

3.5 Translation as an analysis aid

Translation can allow analysis tools that only function for a particular language to analyze code written in other languages. For example, in [15] the authors also take advantage of the conversion from Fortran to C to allow tools that work with C syntax (such as `lint`, a C source linter) to analyse and discover bugs in Fortran programs.

Another example of this usage is the Boogie language [4]. Boogie is an intermediate, language for verification of object oriented programs. Various other programming languages compile to Boogie in order to discharge their proofs, for example, Dafny [18], Chalice [19], and C# via Spec# [5]. An example compilation of Dafny is shown in fig. 3.2.

3.6 Performance of translated code

The performance of translated code is of high importance in all cases where the code is being executed (which is most applications, other than static analysis). In [20] the authors discuss issues with performance when translating COBOL source code to Java. They found that the converted code was 7 to 160 times slower than the source Cobol code. This performance variation can be easily attributed to the overhead introduced when simulated features (for example, in this case, the COBOL data types).

When the languages are similar enough, however, little or no features have to be simulated. In XLT86 [14], for example, due to the relative proximity in features of the source and target language, the overhead is non-existant in many cases.

3.7 Maintainability of generated code

If the goal of the conversion is to migrate out of an old application, library or framework, then it is important that the result of the conversion be maintainable. How readable the generated code is depends mainly on two factors: how many constructs [26] can be directly ported over natively, without simulation, and the quality of the simulations for the constructs that cannot be natively ported over.

As such, language selection plays a large role in how maintainable the generated code can be: If the source and target languages are too dissimilar, then few constructs will be able to be mapped directly. Further, if the target language has does not have the right features, then it can be difficult to find quality simulations. For example, due to the elementariness of assembly languages, it can be difficult to find mappings that maintain readability when high level languages.

3.8 Lowering

A typical step in most traditional compilers (for example, [17]) is the so-called “lowering”, which consists of taking a high level feature and transforming it into lower level code. This is just a particular case of construct simulation, where the high level language is the source language, and the low level language is the target language. Unfortunately, most literature on lowering disregards readability, since the generated code is not for human consumption. As such, most literature is not directly appliable to this work.

```scheme
(print "Hello, world!")
```

(a) Scheme sample.

```c
C_main_entry_point
    void C_ccall
    C_toplevel(C_word c, C_word
     ↪  *av) {
  //19 lines of boilerplate cut
   ↪  out
  a = C_alloc(3);
  C_initialize_lf(lf, 3);
  lf[0] = C_h_intern(&lf[0], 34,
   ↪  C_text("chicken.base"
   ↪  "#implicit-exit-"
   ↪  "handler"));
  lf[1] = C_h_intern(&lf[1], 18,
   ↪  C_text("chicken.base"
   ↪  "#print"));
  lf[2] = C_decode_literal(
   ↪  C_heaptop, C_text(
   ↪  "\376B\000\000\015"
   ↪  "Hello, world!"));
  C_register_lf2(lf, 3,
   ↪  create_ptable());
  {}
  t2 = (*a = C_CLOSURE_TYPE | 2,
   ↪  a[1] = (C_word)f_117, a[2]
   ↪  = t1,
        tmp = (C_word)a, a += 3,
         ↪  tmp);
  {
    C_word *av2 = av;
    av2[0] = C_SCHEME_UNDEFINED;
    av2[1] = t2;
    C_library_toplevel(2, av2);
  }
}
```

(b) Compiled to C.

Figure 3.1: Comparison of source Scheme code and output after compilation by CHICKEN into C.

```
method Mul(x: int, y: int)
 ↪  returns (r: int)
  requires 0 <= x && 0 <= y
  ensures r == x*y
  decreases x
{
  if x == 0 {
    r := 0;
  } else {
    var m := Mul(x-1, y);
    r := m + y;
  }
}
```

(a) Dafny sample.

```
implementation
 ↪  Impl$$_module.__default.Mul(x#0:
 ↪  int, y#0: int) returns (r#0:
 ↪  int, $_reverifyPost: bool)
{
  var $_Frame: <beta>[ref,Field
   ↪  beta]bool;
  var m#0: int;
  var $rhs##0: int;
  var x##0: int;
  var y##0: int;
    $_Frame := (lambda<alpha>
     ↪  $o: ref, $f: Field alpha
     ↪  ::
      $o != null && read($Heap,
       ↪  $o, alloc) ==> false);
    $_reverifyPost := false;
    if (x#0 == LitInt(0))
    {
        r#0 := LitInt(0);
    }
    else
    {
        x##0 := x#0 - 1;
        y##0 := y#0;
        assert (forall<alpha>
         ↪  $o: ref, $f: Field
         ↪  alpha :: false ==>
         ↪  $_Frame[$o, $f]);
        assert 0 <= x#0 || x##0
         ↪  == x#0;
        assert x##0 < x#0;
        call $rhs##0 :=
         ↪  Call$$_module
         ↪  .__default
         ↪  .Mul(x##0, y##0);
        m#0 := $rhs##0;
        r#0 := m#0 + y#0;
    }
}
```

(b) Compiled to Boogie.

Figure 3.2: Comparison of source Dafny code and output after compilation into Boogie.

Chapter 4

Evaluating Candidate Target Languages for Ada Translation

In this chapter, we evaluate various target languages, with various test features, and select one for further development.

4.1 Picking a target language

When picking a target language, one has to keep in mind the extent of overlap of the features of the source and target language. Namely, the subset of features that both languages natively support in the same way is the only features that are going to be perfectly supported (this is the native construct to native construct mapping).

However, this set is often minimal. As such, one also has to keep in mind how the features from the target language are going to be simulated in the source language. The features of the target language can very heavily affect how the source language features can be simulated, and consequently, the readability of the converted code.

4.1.1 Candidate Language Selection

In order to implement the features we want to implement from Ada while following our requirements, we established 5 key features that the ideal target language would be required to have:

Operator overloading Operators are commonplace in programs, either to perform computations or to manipulate indices. Overloading operators for the custom new types defined in the target language allows the programmer to use them as if they were the regular, native integers, and helps keep the converted code readable.

Custom implicit type conversion In order to simulate certain Ada features in a readable manner, it is necessary that the target language permits the programmer to specify custom implicit conversions. They are necessary, for example, to mimic the assignment checking semantics of Ada checked types. Having an explicit conversion between types introduces clutter in the target language code.

Compile-time conversion checking A compiler for the target language should be able to detect whenever type conversions (whether using operator overloading, implicit type conversion or manual conversion) are valid. For example, the C language does not consider it an error when a pointer is cast incorrectly to another, invalid type. A language that does not feature such guarantees adds room for programmer error and requires that all checks be verified during run-time, increasing the overhead.

Metaprogramming support for type generation Programming in Ada-style makes use of a higher number of numerical types than in more mainstream languages. Metaprogramming generally allows the source-code of a program to dynamically modify how the remaining code is processed. In this case, generating the scaffold types (or classes and traits) can be done automatically in the language. A language that does not feature such support requires all used types to be generated into project by a separate tool, making integration harder.

Memory Safety Memory safety is often considered a tradeoff between allowing the programmer to handcraft memory and pointer management for writing highly-performant code and preventing memory-related bugs from occurring. In our case, since the source language is by default memory safe, we also desire the target language to be memory safe.

Considering the most popular languages according to the TIOBE index, plus a few handpicked for their features (see table 4.1), we picked C#, Rust, Scala, and Julia as candidate target languages.

Language	Tiobe Index	Operator Overloading	Implicit Conversions	Compile-time checking	Metaprogramming	Memory safety
Java	1	No	No	Yes	No	Yes
C	2	No	No	Yes	Yes (limited)	No
Python	3	Yes	No	No	Yes	Yes
C++	4	Yes	No	Yes	Yes	No
C#	**5**	**Yes**	**Yes**	**Yes**	**No**	**Yes**
VB.NET	6	Yes	No	Yes	No	Yes
Javascript	7	No	No	No	Yes	Yes
PHP	8	No	No	No	Yes	Yes
SQL	9	?	?	?	?	?
Swift	10	Yes	No	Yes	No	Yes
Rust	**15**	**Yes (limited)**	**No**	**Yes**	**Yes**	**Yes**
Scala	**20**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Julia	**40**	**Yes**	**Yes**	**No**	**Yes**	**Yes**

Table 4.1: Language comparison table.

```
 type Small_Int is range 1 .. 10;
   V : Small_Int := 9;
   W : Small_Int := 2;
begin
   V := V + W - 1;
end;
```

Listing 4.1: Example of intermediate operations outside of range in Ada

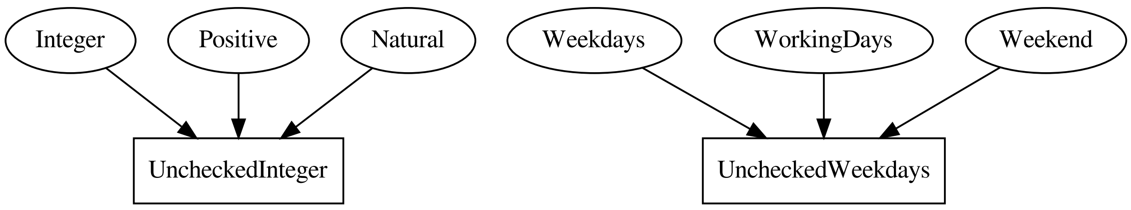

Figure 4.1: Type structure. UncheckedInteger and UncheckedWeekdays represent a type family, while the other nodes represent a specific type of the respective family.

4.1.2 Evaluating potential languages

In order to evaluate the suitability of each language, we set about translating Ada's Ranged and Fixed Point types to candidate languages.

In order to support the different operations across type families, it is necessary to create the corresponding types on the target language. Because Ada allows values to be outside of their allowed ranges in intermediate operations, it is necessary to mimic that state change on the type system of the target language. Figure 4.1 shows an example where `V+W` is outside of the `Small_Int` range, but the final result (`V+W-1`) is inside, so the operation is correct during compilation and execution. Because of this behaviour, there are two kinds of types used in the target language. Each Ada type maps to a unique target type, and it corresponds to values that have to be verified to be inside a bound (e.g. the value being stored in `V`, of type `Small_Int`). For each type family, there is one unchecked type (subtypes can reuse the same type) to represent values during intermediate operations, where no checks occur (e.g. `V+W`). Figure 4.1 show two representative type families, where a single common supertype of unchecked nature is used to represent the family, and all types have a corresponding concrete type (even supertypes, like `Integer`).

When redefining operators on the target language for these custom types, their behaviour should follow Ada semantics: if operands do not share a type family, a static error should be raised, and compilation should be aborted. Just like Ada, all arithmetic operations should check for over and underflow. Since this behaviour is not native, this is a potential source for overheads. Domain constraints, however, are only checked when converting from the general unchecked type to a concrete checked type.

Because of the verbosity and repetition required to implement all of the classes, interfaces, traits, methods and functions required to represent this type structure and associated behavior, metaprogramming techniques can be used to generate them from source-code annotations automatically. Scala, Rust and Julia support such features. Other languages require a scaffolding tool that generates the require infrastructure from a python-based Domain-Specific Language (DSL) describing the Ada-like types. Figure 4.2 shows an example of the DSL used in our examples to generate C# base code. Note that user-code is not generated from this tool.

```
gen_start()
gen_range_type(name="Money", lower=0, upper=100000*100, mul=100,
↪  div=1)
gen_range_sub_type(name="OnlyCents", supertype="Money", lower=0,
↪  upper=99, mul=100, div=1)
gen_end()
```

Figure 4.2: DSL in Python to generate the scaffold classes, structs, functions and methods for C#, which does not support type generation in metaprogramming.

As a running example, we will be using a simple Ada example (Figure 4.3) showing the application of a discount using the `Money` ranged type. Despite being trivial, the example is simple enough to understand the resulting code.

```
function ...
type Money is delta 0.01 range 0.0 .. 100000.0;
Price: Money := 10.5;
quantity : constant := 5;
finalPrice: Money := price * quantity;
begin
finalPrice := finalPrice * 0.70;
end ...
```

Figure 4.3: Ada version of discount program.

4.1.3 Rust

Rust is a strongly typed programming language initially designed by Mozilla to help programmers catch concurrent bugs during compilation. As such, Rust features a borrow-checker that detects aliasing and memory problems. Performance-wise, Rust aims to achieve results as good as C, since the target applications are the same.

Ada's types are implemented using one struct per type family and one struct per concrete type (as seen in Figure 4.4). Since Rust structs do not have behaviour, conversions and operations are implemented between each pair of types that can be converted directly.

Since there is no generic support in Rust at the moment, code size increases exponentially to the number of types in a type family. Rust procedural macros can be used to generate these types from the source code, not requiring any external tool.

```rust
pub struct UncheckedInteger(i64);
pub struct Integer(i64);
```

Figure 4.4: Generated Rust structs

```rust
impl From<UncheckedInteger> for Integer { /* ... */ }
```

Since Rust doesn't have custom automatic conversions, manual conversions (and consequently, manual checking) must be used. These are implemented using the native `From<T>` trait. In addition, the rust type inference is not enough to detect all correct intermediate types, so type annotations or auxiliary are often needed (e.g. Figure 4.5).

```rust
let mut price: Money = (10.5).into();
let quantity = 5;
let tmp: Money = quantity.into();
let final_price: Money = (price * tmp).into();
```

Figure 4.5: Partial Rust translation of discount program

4.1.4 C#

C# is a general-purpose programming language that is used to write a wide range of software, from device drivers to mobile applications. For performance reasons, structs were used to implement types instead of classes, where the dispatching overhead would be significant.

Each concrete type is represented by a struct like in Rust, but two structs are now required per type family: one to represent an unchecked value and a second type is required to be used as a parameter in generic code to identify classes as members of the type family. While this struct has no content nor is it associated with any behavior, it exists only to enforce subtyping.

Figure 4.6 shows an example of this structure, where `UncheckedAIntegerType` is the empty type marker, only used to unify other types (`UncheckedAInteger` and `Ainteger` and, automatically, their subtypes) under the same type in C#.

Implicit conversions are used to represent conversion between valid (i.e. under the same type family) types. Figure 4.7 shows the implicit conversions of the previous example.

These implicit conversions allow the resulting code (in Figure 4.8) to be very similar (except syntactic language differences) to the original Ada discount example.

```
public readonly struct UncheckedAIntegerType { }
public readonly struct UncheckedAInteger :
↪  IntValued<UncheckedAIntegerType> { /* ... */ }
public readonly struct AInteger : IntValued<UncheckedAIntegerType>
↪  { /* ... */ }
```

Figure 4.6: Generated C# structs. AInteger is used instead of Integer in order to avoid name clashing with the native .NET Integer.

```
public static implicit operator AInteger(long i) => new
↪  AInteger(i);
public static implicit operator AInteger(UncheckedAInteger i) =>
↪  new AInteger(i.Value);
```

Figure 4.7: C# implicit conversions

4.1.5 Scala

The Scala language was designed to execute on top of the JVM and, as such, shares much of its design. JVM custom numeric types (such as `BigInteger` or `BigDecimal`) are designed using classes and objects and thus are inefficient due to instantialization and a large amount of garbage collector pressure.

Similarly to how Scala handles types representing native types (`scala.Integer`), we leverage Scala value classes to elide objects and implement the required operations with no overhead.

Unlike C#, Scala does not require the additional type marker, thus requiring only one class per concrete type and one class per type family to represent unchecked behavior. Figure 4.9 shows the type structure in Scala for the same example.

Similarly to C#, Scala also implements implicit conversions that remove the need for Rust's `.into()` calls. However, unlike C#, Scala supports macros, removing the need to generate classes in a separate compilation step.

However, even with implicit conversions, the Scala type system considers operators as methods on the left-side object, using the right-side object as the argument. In fact, `1 + a` is the same as `1.+(a)`. This Scala behavior prevents a clean implementation of arithmetic operators on custom types that seem entirely native. When the left object is an instance of the custom type, the operator method is called on the right class. However, when the left object is a native `scala.Int` and the right is a custom type, there is no viable method in the Scala native class suitable.

To work around this behavior, an alternative operator is available for all native types under a slightly different name that has a ! prepended. Figure 4.10 presents the discount example in Scala, with no major differences to Ada. Figure 4.11 shows two correct, but

```
Money price = 10.5;
var quantity = 5;
Money finalPrice = price * quantity;
finalPrice *= 0.70;
```

Figure 4.8: C# translation of discount program

```
class UncheckedInteger(val value: Long) extends AnyVal { /* ... */ }
class Integer(val value: Long) extends AnyVal { /* ... */ }
```

Figure 4.9: Generated Scala classes

syntactically different, alternatives for the discount example, as well as an incorrect one.

```
val price: Money = 10.5
val quantity = 5
var finalPrice: Money = price * quantity
finalPrice *= 0.70
```

Figure 4.10: Scala translation of price program

4.1.6 Julia

Julia is a language targeted at scientific applications, and its nature makes it very distinct from the previous languages. In fact, Julia has more similarities with Python, Javascript or Perl in its feature range. While Rust is a compiled language and C# and Scala are hybrid languages that are compiled to a bytecode that gets interpreted and eventually JIT-compiled. Despite having no syntax similarities, Julia is admittedly a LISP, having full hygienic macros and being homoiconic. The irony is that Julia takes advantage of those features to implement a very advanced type system[1], used to specialize functions right before they are executed, in order to have a very concrete version of the function for the particular argument types.

The translation of Ada types to Julia also takes advantage of these features: hygienic macros are used to automatically generate the type structure that represents Ada's types (Figure 4.12). The translation to Julia is unique in this aspect, not requiring an external tool to generate source code. This is a very important distinction (and the reason to select a very dynamic language like Julia) because having type declarations directly in the source code allows for dynamic and interactive creation and editing of types. Doing so in other languages requires editing the DSL file and re-running the tool.

[1]dynamic, nominative and parametric.

```
finalPrice = finalPrice * 0.7 // Correct
finalPrice = 0.7 * finalPrice // Error
finalPrice = 0.7 !* finalPrice // Correct Alternative
finalPrice = finalPrice !* 0.7  // Also Correct
```

Figure 4.11: Alternative approaches for the discount example in Scala.

```
@fixed_point_type(0, 100000*100, 100, 1, Money)
@fixed_point_subtype(0, 99, OnlyCents, Money)
```

Figure 4.12: Julia macro-based API example

Like in the other languages, the advanced type system of Julia is used to limit the allowed operations to the correct Ada semantics. All of these types have a minimal impact on the execution, as only the native Julia types are used in the specialization and optimization of the function during JIT compilation.

Type representation in Julia relies on abstracts types to represent type families. For each concrete type, a Julia struct is created, subclassing the `Ranged{T}` type parameterized with the respective type family. A single `UncheckedRanged` struct is required to represent all unchecked values from all type families (unlike in previous languages). This struct is parametrized with the concrete type and regardless of the type parameter, it is a subtype of the native Integer type. (Fig. 4.13).

```
abstract type AdaIntType <: RangedType end
struct AdaInt <: Ranged{AdaIntType} #= ... =# end
struct UncheckedRanged{T<:RangedType} <: Integer #= ... =# end
```

Figure 4.13: Generated Julia types and structs

Similarly to other languages, Julia supports implicit conversions (called promotions), but a single generic rule is enough to cover all types (Fig. 4.14). Not only there is no exponential complexity in the number of rules, but types can also be much effortlessly added or removed.

The heavy meta-programming features of Julia allow the resulting code to look very similar to the original Ada code (Fig. 4.15), with types being defined in the same source file.

4.2 Discussion

From an ergonomics point of view, Ada can only be fully expressed in Julia, with other languages having some kind of barrier (lack of macros for type generation, exponential

```
promote_rule(::Type{T}, ::Type{U}) where
 ↪  {V,T<:Union{UncheckedRanged{V},Ranged{V}},U<:Integer} =
 ↪  UncheckedRanged{V}
```

Figure 4.14: Julia promotion implementation

```
price::Money = 10.5
quantity = 5
finalPrice::Money = price * quantity
finalPrice *= 0.70
```

Figure 4.15: Julia translation of price program

behavior to implement conversions, operator overloading limitations). However, Julia was not designed to be a systems language, despite its advanced type system. Rust shares procedural macros with Julia, also allowing source-level type generation. However, Rust has very limited automatic type conversion, which introduces unnecessary artefacts on the source code.

C# both requires an extra tool to generate the corresponding types, but other than that presents a very elegant idiomatic integration.

Scala is almost equivalently idiomatic (except for the operator-method behaviour described in listing 5.5).

To evaluate the overhead of the translation for each language, we designed a new benchmark that iteratively repeats arithmetic operations on ranged types. The goal of this benchmark is to evidence the overhead introduced by checking operations on all languages. The benchmark compares the same operations on the native unchecked type against the checked version introduced to represent types.

The benchmark consists of repeatedly incrementing two variables (which variable is incremented is randomly chosen to prevent overeager optimization) 10^8 times.

The benchmark was executed on a Windows 10 machine with an Intel(R) Core(TM) i7-7700HQ 4 core 2.80GHz CPU. Table 4.2 shows the versions used for each language implementation.

Language	Version
Rust	rustc 1.36
Java	Java HotSpot(TM) 64-Bit Server VM (build 25.231-b11, mixed mode)
C#	Visual C# 2019
Julia	julia 1.2.0

Table 4.2: Language versions benchmarked

Table 4.3 presents the results of the benchmark, showing that Scala was the language with the least overhead, mainly due to an already expensive native performance. Julia and Rust both provide a good absolute time on both unchecked and checked versions, which makes them suitable for performance-critical applications.

Language	Unchecked (ms)	Checked (ms)	Overhead (%)
Rust	73.3	310.4	76.39
Scala	264.6	312.4	15.29
Julia	130.4	309.2	57.81
C#	303.8	667.2	54.47

Table 4.3: Overhead of using ranged types over non-checked integers in an arithmetic-intensive benchmark.

Regarding the choice of language from conversion, Rust, Scala and Julia produces the fastest executables. However, different languages have different tradeoffs: Rust will produce native executables; However, it requires manual casts. On the other hand Julia and Scala source end up looking quite close to Ada source, however, in Julia errors can only be detected at run-time and in Scala some small workarounds are needed. And C#, which is the slowest, has the best source code, with no workarounds required.

After weighing all these factors, we picked Scala as our target language.

Chapter 5

Ada to Scala translation

The compiler consists of a python script that operates directly on the abstract sytax tree (AST) of the Ada code (obtained via `libadalang` [3]), generating a semantically equivalent Scala AST, as shown in fig. 5.1. The conversion happens via several various visitor-like objects. Of note is that one Ada node does not necessarily correspond to a single Scala node because most features require a more complex translation strategy. In addition, even when the AST nodes match one-to-one, in many cases complex support libraries using advanced Scala features had to implemented. This chapter addresses the challenges of these more complex conversions.

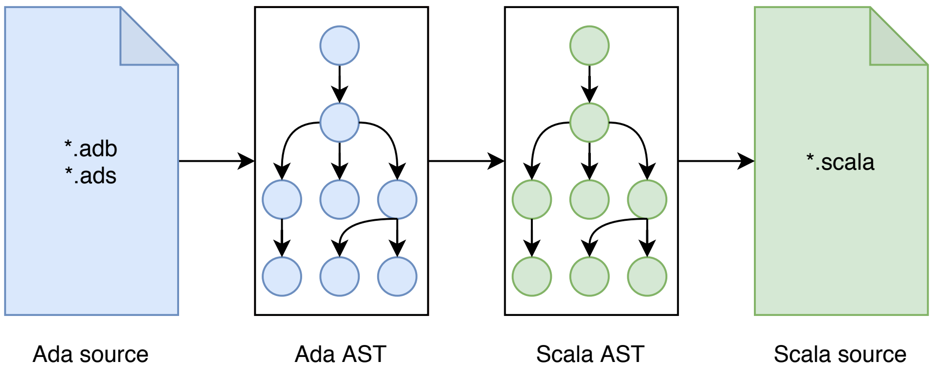

Figure 5.1: Functioning of the compiler.

5.1 Syntax swap

The syntax swap part of the conversion is done implicitly, by emitting the corresponding Scala AST Node for the Ada node being visited. For example, Ada attribute accessor nodes which use single quotes (`Type'Attribute`) get converted one-to-one into Scala accessor nodes which use dots (Type.Attribute). In addition, Ada files are converted into objects, procedures into functions returning Unit (also known as void in other languages), and the main method is added that bootstraps the Task system and runs the correct main function.

5.2 Translation of checked and fixed-point types

Checked and fixed-point types pose a difficulty in translation to Scala. One way to translate these would be to create a simple class representing each type and force the usage of methods to perform operations. For example `(a * b) + (c * d)` might get translated to `a.mul(b).add(c.mul(d))`. This approach has the disadvantage of heavily harming readability due to the verbosity of the generated code. In order to circumvent this, one might think to use operator overloading.

Using this scheme, one might think to put overflow checking code on the operator methods. However, let's take at the behaviour of Ada with the code in listing 5.1. While the value of the `a + b` subexpression goes out of bounds, the code still compiles and runs. Let us explore why this behaviour happens: In Ada, bounds are only checked when the value is assigned to a variable (or passed as a parameter). This means that a subexpression can temporarily escape the permitted values, as long as the whole expression falls back in bounds before being assigned.

```ada
type Money is delta 0.01 range 0.0 .. 1000.0;
A : Money = 900.0;
B : Money = 500.0;
Average : Money = (A + B) / 2; -- Temporary overflow is OK!
```

Listing 5.1: Demonstration of the subexpression overflow behaviour.

If bounds checking is done on each arithmetic operation, this behaviour is not reproduced. Instead, we must have two classes of values: checked and unchecked. However, we must now answer another relevant question: how exactly does the conversion from unchecked to checked happen? One might consider doing it manually like the following: `C = ((A + B) / 2).check()`. However, again, this quickly becomes unwieldy when there is code with lots of assignments. Even worse, when modifying the code after conversion, it's very easy to accidentally forget to check the result, which would make the code lose some safety properties.

Instead, ideally, we would like to override the assignment operator to perform the checking. In Scala, this is not possible. Instead, we define an implicit function that converts from the Unchecked version to the Checked variant. A call to this function is then inserted by the compiler every time the compiler needs the right side of the assignment to match the type of the left side.

With this, we convert operations on types without introducing any syntatic overhead when compared to the Ada version. For example, the code in listing 5.3 gets translated to the Scala code in listing 5.4.

However, the Scala type system considers operators as methods on the left-side object, using the right-side object as the argument. In fact, `1 + a` is the same as `1.+(a)`. When

```
implicit def checkUncheckedIntegerToInteger(unchecked:
↪ UncheckedInteger): Integer = { /* ... */ }
implicit def uncheckInteger(value: Integer): UncheckedInteger =
↪ { /* ... */ }
```

Listing 5.2: Implementation of checking and unchecking using Scala implicit functions.

```
function Discount
type Money is delta 0.01 range 0.0 .. 100000.0;
Price: Money := 10.5;
quantity : constant := 5;
finalPrice: Money := price * quantity;
begin
finalPrice := finalPrice * 0.70;
end Discount
```

Listing 5.3: Original Ada discount program.

the left object is an instance of our custom type, the operator method is called on the right class. However, when the left object is a native `scala.Int`, the compiler finds the built-in addition method. Then, when the right object is our custom type, the compiler skips implicit method resolution and simply throws up an error. In order to work around this issue, an alternative operator is available under a slightly different name that has a ! prepended. Listing 5.5 showcases how this bug manifests.

5.3 Translation of Tasks

Another feature that is unique to Ada, posing a difficulty in translation is its task-based concurrency system.

Firstly, since each task can run independently, it is natural to map each task to a thread. This is the approach we took. However, there is quite a bit of functionality (that needs to be built on top of simple threads) that is common for all tasks. In order to take advantage of this, and produce cleaner code, we developed an internal DSL that allows succinctly expressing tasks in Scala source. We firstly present how a task gets converted in-source. Then we will present how the macros expand. Finally we will present how the runtime libraries are implemented.

5.3.1 Task Conversion

Firstly, each task gets converted into an anonymous subclass of the Task class. Each task entry is wrapped with a custom macro, explained later in section section 5.3.2. For now we will assume it makes the case class have the same behavior as an Ada entry.

```
@FixedKind(100, 1)
@Fixed(0*100, 100000*100)
class Money

val price: Money = 10.5
val quantity = 5
var finalPrice: Money = price * quantity
finalPrice *= 0.70
```

Listing 5.4: Scala translation of discount program.

```
finalPrice = finalPrice * 0.7 // Compiles correctly
finalPrice = finalPrice !* 0.7  // Also compiles correctly
finalPrice = 0.7 * finalPrice // Doesn't compile!
finalPrice = 0.7 !* finalPrice // Correct alternative
```

Listing 5.5: Alternative approaches for the discount example in Scala.

Standalone `accept` statements are converted to calls to the `accept_one` function. `accept` statements inside `select` statements are converted to `accept` functions inside `select` calls. The accept function takes two callbacks: the synchronous part, which gets ran while the calling task is locked and the asynchronous part, which is ran after it gets unlocked.

Calls to entries get converted to standard scala calls on objects.

An example conversion of a sample from listing 2.7 is shown in listing 5.7.

5.3.2 The Entry Macro

The `@Entry` macro is used to improve the readability of generated code. An example expansion is shown in 5.6. The expanded definition's apply method[1] is used to allow directly calling entries, without having to instantiate the class first.

```
case class Increment(Value: Integer) extends this.OpClass
object Increment { def apply(Value: Integer): Unit = runOp(new
↪ Increment(Value)) }
```

Listing 5.6: Expansion of the Entry macro as applied to the Increment class in listing 5.7 in lines 9 and 10.

[1]The apply method of an object or class is used to specify what happens when the object or class is treated as a function. For example, if we have an object `a`, that object's apply method specifies what happens when `a()` is called.

```scala
import Ada.{Text_IO => TIO}
import Ada.{Integer_Text_IO => ITIO}
import adascala.prelude._
object P {
  def P(): Unit = {
    val T = new Task {
      @Macros.Entry
      case class Print()
      @Macros.Entry
      case class Increment(Value: Integer)
      override def runTask(): Unit = {
        var CurrValue: Integer = 0L
        loop {
          select(
            accept[Print](
              { case Print() if CurrValue % 2L == 0L => },
              () => {
                ITIO.Put(CurrValue)
                TIO.Put_Line(" is even")
              }
            ),
            accept[Print](
              { case Print() if CurrValue % 2L != 0L => },
              () => {
                ITIO.Put(CurrValue)
                TIO.Put_Line(" is odd")
              }
            ),
            accept[Increment]({
              case Increment(_Value: Integer) =>
                CurrValue = CurrValue + _Value
            }),
            terminate
          )}}}
    T.Print()
    T.Increment(3L)
    T.Print()
    T.Increment(7L)
    T.Print()
  }
  def main(args: Array[String]): Unit = {
    Task.bootstrapMain(args, () => { P() });
  }
}
```

Listing 5.7: Scala translation of 2.7

5.3.3 Task Runtime

The runtime part of the task implementation consists of the Task class, which provides a wrapper around native scala threads. Each task contains a queue for each type of entry. When a task starts accepting an entry, it monitors the queues for the appropriate entry and sleeps if needed. When an appropriate entry is invoked, the task wakes up, and in sequence, first executes the synchronous block, then unlocks the caller, and finally runs the asynchronous block.

In adition, in program entry points, a call is added to `Task.bootstrapMain(...)` in order to initialize the Task runtime.

5.4 Translation of Parameters

Ada parameters can have in and out specifiers which indicate if writes to the parameter inside the function or entry statement will propagate outside. In Scala, there is no pass-by-reference, which is the most normal way to implement this. We now show how we can cleanly implement this feature. Firstly, Scala allows anonymous functions to modify variables local to the enclosing function. As such, in order to implement an out parameter, we can pass a callback that writes to the variable that is supposed to be out (like the following: `x => variable = x`, full example in listing 5.8). However, to maximize readability and similarity with the source code, we would like to use out parameters precisely the same as in parameters. In order to implement this, we firstly define three wrapper types: `In[T]`, `Out[T]` and `InOut[T]`. Now, when we pass in a parameter, we just pass in a variable of type T, creating a mismatch in the types. We can abuse this to create a custom implicit conversion that generates the appropriate callback. However, a standard implicit conversion would just receive a copy of the value, which would invalidate the callbacks. Thankfully, scala supports implicit macros, and as such, our implicit conversion function can be a macro, completing the puzzle.

A full example taking advantage of all types of parameters is shown in listing 5.9.

5.5 Translation of Arrays

Arrays are one of the most common data structures in programs. This fact makes it superlatively important that boilerplate is kept to a minimum.

As previously hinted, Ada arrays differ from Scala arrays in a few major ways, namely:

- Ada arrays can start at any index, Scala arrays start at 0.
- Ada supports slicing, Scala doesn't
- Ada has a terse syntax for array initialization and copy, Scala doesn't.

```scala
object ByReference {
  def assignFour(variable: Integer => Unit): Unit = {
    variable(4) // Assign by calling the function
  }

  def main(args: Array[String]): Unit = {
    var variable = 0;
    assignFour(value => variable = value)
    println(variable)
  }
}
```

Listing 5.8: Pass by reference simulated in scala. This example prints “4”.

We implement all arrays as objects of a custom class AdaArray. This class supports the 3 aforementioned points.

Firstly, to support starting at any index, we simply remap indexes from the range exposed to the programmer, to the range actually supported by the backing array. This is done by subtracting the value of the first index from the exposed indexes.

Secondly, we supports slicing via the creation of objects that provide different “views” into the same backing array.

Finally, we do all of this in a terse way via overriding the `apply` and `update` methods, which allow our new array to be just as terse as a native array.

Translations of the two examples of array usage previously presented are shown in listing 5.10 and listing 5.11.

5.6 Translation of Generics

Generics are essential for the implementation of generic data structures. Firstly, generics on functions are translated in a straight forward manner: generic parameters for generic functions in Ada are transformed into generic parameters for generic functions in Scala (as shown in listing 5.12).

However, when it comes to parameterizing a block of functions, Ada is more flexible, due to the fact that one can parameterize a block of functions at once via a package declaration. In Scala only classes and methods can be parameterized. The Scala equivalent (an object declaration) cannot be parameterized. To work around this issue, we instead use a class declaration as shown in listing 5.13. Instantiating the package then becomes an instantiation of the class.

In addition, Scala only allows generic parameters to be classes. As such, since Ada allows generic parameters to be anything (they can be, for example, functions), in situations where these are used, an alternative is used. For a generic method, parameters are added,

```scala
import Ada.{Integer_Text_IO => ITIO, Text_IO => TIO}
import adascala.prelude._

object InOutTest {
  def main(args: Array[String]): Unit = {
    var a: Integer = 10;
    print(a)
    set(a)
    print(a)
    add(a)
    print(a)
  }

  //In
  def print(a: Integer): Unit = {
    ITIO.Put(a)
    TIO.New_Line()
  }

  def set(a: Out[Integer]): Unit = {
    a() = 100
  }

  def add(a: InOut[Integer]) {
    a() = a() + 10
  }
}
```

Listing 5.9: Full In and InOut parameter example. Prints “10”, “100”, “110”.

and the function is curried, such that the function can be partially applied with the "generic" parameters. For a generic class, the parameters are added to the constructor, and similarly curried.

```scala
import Ada.{Integer_Text_IO => ITIO}
import adascala.prelude._
object A {
  @Macros.Ranged(-1L, 1L)
  class R
  def A(): Unit = {
    val Arr = new AdaArray[Integer](R.lower, R.upper)
    Arr() = AdaArray(1L, 0L -> 0L)
    Arr() = AdaArray(0L, 0L -> 1L)
    ITIO.Put(Arr(-1L))
    ITIO.Put(Arr(0L))
    ITIO.Put(Arr(1L))
  }
  def main(args: Array[String]): Unit = {
    Task.bootstrapMain(args, () => { A() });
  }
}
```

Listing 5.10: Scala translation of array example in listing 2.9. This example prints “0”, “1”, “0”.

```scala
import Ada.{Integer_Text_IO => ITIO}
import adascala.prelude._
object A {
  @Macros.Ranged(-1L, 1L)
  class R
  def A(): Unit = {
    val Arr = new AdaArray[Integer](R.lower, R.upper)
    Arr(0L) = 1L
    Arr(1L) = 2L
    Arr(-1L to 0L) = Arr(0L to 1L)
    ITIO.Put(Arr(-1L))
    ITIO.Put(Arr(0L))
    ITIO.Put(Arr(1L))
  }
  def main(args: Array[String]): Unit = {
    Task.bootstrapMain(args, () => { A() });
  }
}
```

Listing 5.11: Scala translation of array example in listing 2.10. This example prints “1”, “2”, “2”.

```ada
generic
  type Element_T is private;
procedure Swap (X, Y : in out Element_T);
```

(a) Header in Ada.

```scala
def Swap[Element_T](_X: InOut[Element_T], _Y: InOut[Element_T]):
 ↪  Unit
```

(b) Translated to Scala.

Listing 5.12: Generic function header in Ada and Scala.

```ada
-- .ads
generic
   type T is private;
package Element is
   procedure Set (E : T);
   function Get return T;
private
   Value : T;
end Element;
-- .adb
package body Element is
   procedure Set (E : T) is
   begin
      Value := E;
   end Set;
   function Get return T is
   begin
      return Value;
   end Get;
end Element;
```

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Element;
procedure Show_Generic_Package is
   package I is new Element (T => Integer);
begin
   Put_Line ("Initializing...");
   I.Set (5);
   Put_Line ("Value is now set to " & Integer'Image
   ↪  (I.Get));
   I.Set (3);
   Put_Line ("Value is now set to " & Integer'Image
   ↪  (I.Get));
end Show_Generic_Package;
```

(a) Header in Ada.

```scala
import adascala.prelude._
class Element[T] {
  var Value: T = 0L
  def Set(_E: T): Unit = {
    Value = _E
  }
  def Get(): T = {
    return Value
  }
}
```

```scala
import Ada.{Text_IO => TIO}
import adascala.prelude._
object Show_Generic_Package {
  def Show_Generic_Package(): Unit = {
    var I = new Element[Integer]()
    TIO.Put_Line("Initializing...")
    I.Set(5L)
    TIO.Put_Line("Value is now set to " + Integer.Image(I.Get))
    I.Set(3L)
    TIO.Put_Line("Value is now set to " + Integer.Image(I.Get))
  }
  def main(args: Array[String]): Unit = {
    Task.bootstrapMain(args, () => { Show_Generic_Package() });
  }
}
```

(b) Translated to Scala.

Listing 5.13: Generic package demo in Ada and Scala.

Chapter 6

Evaluation

We evaluate our results based on the number of convertible features and an empirical survey on the readability of the converted code.

In this chapter, we present the test programs and their translations which were included in the survey. We then present the methodology and results of the survey.

6.1 Example programs

The whole work was continuously evaluated with test programs that are representative of the features implemented. In the following subsections, we will present the subset of the test programs and their translations that were used in the survey.

6.1.1 Hello World

This example, shown in listing 6.1 showcases one of the most straightforward possible conversions, and demonstrating elements that are common to every conversion.

```ada
with Ada.Text_IO;

procedure Hello is
begin
   Ada.Text_IO.Put_Line ("Hello, world!");
   Ada.Text_IO.Put_Line ("Hello, world two!");
end Hello;
```

(a) Ada sample.

```scala
import adascala.prelude._
object Hello {
  def Hello(): Unit = {
    Ada.Text_IO.Put_Line("Hello, world!")
    Ada.Text_IO.Put_Line("Hello, world two!")
  }
  def main(args: Array[String]): Unit = {
    Task.bootstrapMain(args, () => { Hello() });
  }
}
```

(b) Compiled to Scala.

Listing 6.1: Hello World sample.

In general, files are converted into objects, procedures into functions returning Unit (also known as void in other languages), and a main method is added that bootstraps the environment and runs the correct main function.

6.1.2 Ranged type

The next example, shown in listing 6.2 shows how ranged data types are converted. In Ada, these data types are ubiquitous, and as such, it is important that translation be as simple as possible.

```
procedure RTest is
   type Range_Type is range -5 .. 10;
   subtype Sub_Range is Range_Type range 0 .. 2;
   Result : Range_Type := 8;
begin
   Result := Result + Result - Result;
   Result := Result + Result; -- Runtime Error here
end RTest;
```

(a) Ada sample.

```
import adascala.prelude._
object RTest {
  @Macros.Ranged(-5L, 10L)
  class Range_Type
  @Macros.Ranged(0L, 2L)
  class Sub_Range extends Range_Type
  def RTest(): Unit = {
    var Result: Range_Type = 8L
    Result = Result + Result - Result
    Result = Result + Result // Runtime error here
  }
  def main(args: Array[String]): Unit = {
    Task.bootstrapMain(args, () => { RTest() });
  }
}
```

(b) Compiled to Scala.

Listing 6.2: Ranged Datatype sample.

6.1.3 Greatest Common Divisor

This sample, shown in listing 6.3 demonstrates loops, function calls and attributes.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Gcd_Test is
   function Gcd (A, B : Integer) return Integer is
      M : Integer := A;
      N : Integer := B;
      T : Integer;
   begin
      while N /= 0 loop
         T := M;
         M := N;
         N := T mod N;
      end loop;
      return M;
   end Gcd;
begin
   Put_Line ("GCD of 100, 5 is"
      & Integer'Image (Gcd (100, 5)));
   Put_Line ("GCD of 5, 100 is"
      & Integer'Image (Gcd (5, 100)));
   Put_Line ("GCD of 7, 23 is"
      & Integer'Image (Gcd (7, 23)));
end Gcd_Test;
```

(a) Ada sample.

```
import Ada.{Text_IO => TIO}
import adascala.prelude._
object Gcd_Test {
  def Gcd_Test(): Unit = {
    def Gcd(_A: Integer, _B: Integer): Integer = {
      var M: Integer = _A
      var N: Integer = _B
      var T: Integer = 0L
      while (N != 0L) {
        T = M
        M = N
        N = T % N
      }
      return M
    }
    TIO.Put_Line("GCD of 100, 5 is" + Integer.Image(Gcd(100L,
    ↪  5L)))
    TIO.Put_Line("GCD of 5, 100 is" + Integer.Image(Gcd(5L,
    ↪  100L)))
    TIO.Put_Line("GCD of 7, 23 is" + Integer.Image(Gcd(7L,
    ↪  23L)))
  }
  def main(args: Array[String]): Unit = {
    Task.bootstrapMain(args, () => { Gcd_Test() });
  }
}
```

(b) Compiled to Scala.

Listing 6.3: Greatest Common Divisor sample.

The translations are straightforward: loops into loops, functions into functions, and attribute calls into object calls.

6.1.4 Tasks

The next two translations deal with tasks and are shown in listings 6.4 and 6.5. The first shows how a minimal task can be started. The second shows how task entries and accept statements get converted.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Task_Print_Demo is
   task PrintTask;
   task body PrintTask is
   begin
      Put_Line ("Printing from print task");
   end PrintTask;
begin
   Put_Line ("Printing from main task");
end Task_Print_Demo;
```

(a) Ada sample.

```
import Ada.{Text_IO => TIO}
import adascala.prelude._
object Task_Print_Demo {
  def Task_Print_Demo(): Unit = {
    val PrintTask = new Task {
      override def runTask(): Unit = {
        TIO.Put_Line("Printing from print task")
      }
    }
    TIO.Put_Line("Printing from main task")
  }
  def main(args: Array[String]): Unit = {
    Task.bootstrapMain(args, () => { Task_Print_Demo() });
  }
}
```

(b) Compiled to Scala.

Listing 6.4: Simple printing task comparison.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure S is
   task PrintTask is
      entry Sync;
   end PrintTask;
   task body PrintTask is
   begin
      Put ("1");
      accept Sync do Put ("2"); end;
      Put ("3");
   end PrintTask;
begin
   Put ("1");
   PrintTask.Sync;
   Put ("3");
end S
```

(a) Ada sample.

```
import Ada.{Text_IO => TIO}
import adascala.prelude._
object S {
  def S(): Unit = {
    val PrintTask = new Task {
      @Macros.Entry
      case class Sync()
      override def runTask(): Unit = {
        TIO.Put("1")
        accept_one[Sync]({
          case Sync() =>
            TIO.Put("2")
        })
        TIO.Put("3")
      }
    }
    TIO.Put("1")
    PrintTask.Sync()
    TIO.Put("3")
  }
  def main(args: Array[String]): Unit = {
    Task.bootstrapMain(args, () => { S() });
  }
}
```

(b) Compiled to Scala.

Listing 6.5: Simple syncronizing task comparison.

6.1.5 Record

This example shows how records are translated. Records are an essential feature in Ada, grouping together several variables together, similar to a C struct.

6.1.6 Out Parameter

This sample, in listing 6.7 demonstrates how out parameters are translated. During the translation, the keywords `OUT` and `IN OUT` are replaced by a wrapper type that indicates the direction of data flow.

```ada
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Record2 is
   type Date is record
      Day : Integer range 1 .. 31;
   end record;
   A : Date := (Day => 1);
   B : Date := (Day => 2);
begin
   Put (A.Day, 2);
   Put (B.Day, 2);
   A := B;
   Put (A.Day, 2);
   Put (B.Day, 2);
   B.Day := 3;
   Put (A.Day, 2);
   Put (B.Day, 2);
end Record2;
```

(a) Ada sample.

```scala
import Ada.{Text_IO => TIO}
import Ada.{Integer_Text_IO => ITIO}
import adascala.prelude._
object Record2 {
  def Record2(): Unit = {
    case class Date(var Day: Integer = 0L)
    var A: Date = new Date(Day = 1L)
    var B: Date = new Date(Day = 2L)
    ITIO.Put(A.Day, 2L)
    ITIO.Put(B.Day, 2L)
    A = B.copy()
    ITIO.Put(A.Day, 2L)
    ITIO.Put(B.Day, 2L)
    B.Day = 3L
    ITIO.Put(A.Day, 2L)
    ITIO.Put(B.Day, 2L)
  }
  def main(args: Array[String]): Unit = {
    Task.bootstrapMain(args, () => { Record2() });
  }
}
```

(b) Compiled to Scala.

Listing 6.6: Record sample.

```ada
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure A is
   procedure B (O : out Integer) is
   begin
      O := 2;
   end B;
   V : Integer := 0;
begin
   B (V);
   Put (V);
end A;
```

(a) Ada sample.

```scala
import Ada.{Integer_Text_IO => ITIO}
import adascala.prelude._
object A {
  def A(): Unit = {
    def B(_O: Out[Integer]): Unit = {
      _O() = 2L
    }
    var V: Integer = 0L
    B(V)
    ITIO.Put(V)
  }
  def main(args: Array[String]): Unit = {
    Task.bootstrapMain(args, () => { A() });
  }
}
```

(b) Compiled to Scala.

Listing 6.7: Out Parameter sample.

6.2 Survey

In order to qualitatively evaluate the translations, we conducted a survey from a pool of Ada and Scala programmers. In this section we will present and analyse the results of the survey. The participants were shown a sample of code, its translation (as done by our prototype), and asked to rank the translations in three axis: similarity to the original Ada code (henceforth referred to as *similarity*), similarity to regular handwritten Scala code (*nativeness*) and general readability (*readability*).

Participants were gathered from the Scala and Ada *subreddits* and the `https://users.scala-lang.org/` forum. There were 25 participants. Of these, 14 reported some familiarity with Ada, while 19 reported some familiarity with Scala.

The results of the survey are shown in figs. 6.1 to 6.3. From these results we can see that most samples have a *similarity* of around 4, a *nativeness* between 2 and 3, and *readability* of around 3.

However, there are two tests that stand out negatively from the others in their *similarity* and *readability* scores: the Task (sync) and Range types tests. The Task (sync) test utilizes tasks and task entries and performs syncronization. The Range test defines a new ranged datatype and performs some arithmetic with it. Coincidentally these are the features that are most unique to Ada, and thus are most challenging to translate due to having no equivalent in Scala. Therefore, it is natural that the translations seem more unorthodox or less readable: the only way to implement these features completely cleanly would be to have a language especially tailored for this purpose.

Conversely, two tests that stand out positively: Hello World and Greatest Common Divisor. Hello World simply prints hello word, while the Greatest Common Divisor defines a function that uses a loop to calculate the Greatest Common Divisor, and invokes it. These two tests use no Ada features that are not directly present in Scala, and as such, are trivial to translate, thereby obtaining the best scores.

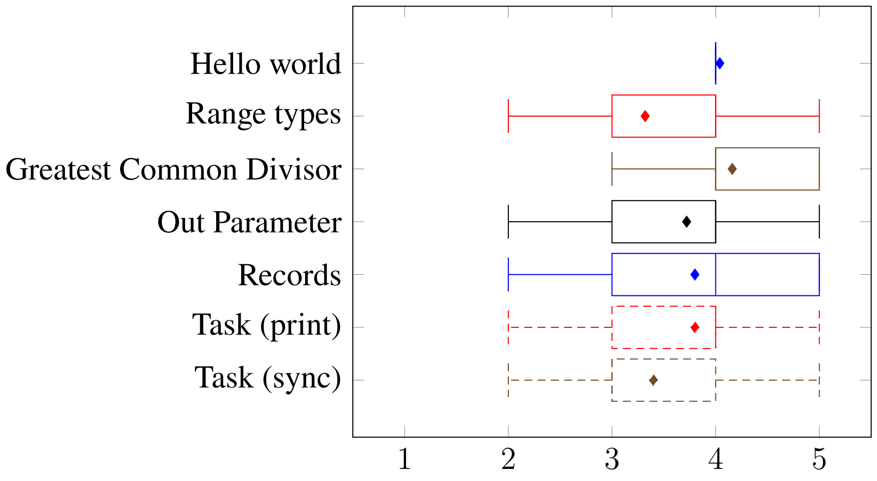

Figure 6.1: Results of similarity to the original Ada code

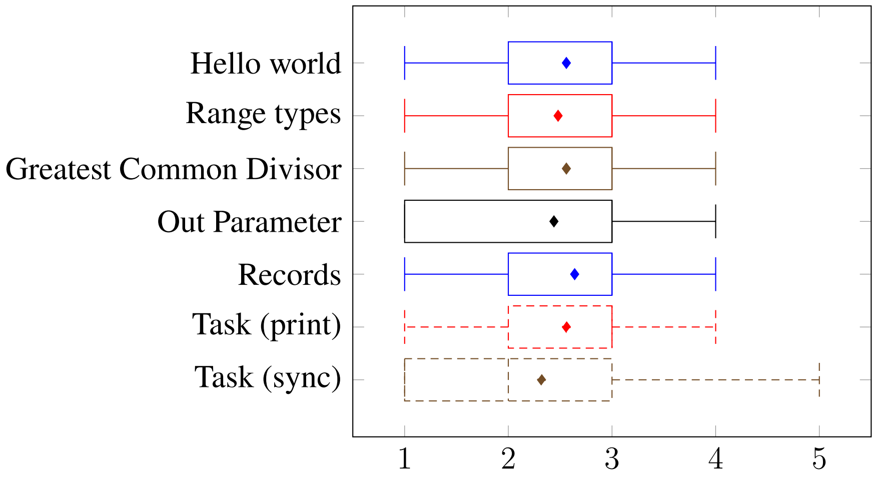

Figure 6.2: Results of similarity to regular, handwritten Scala code

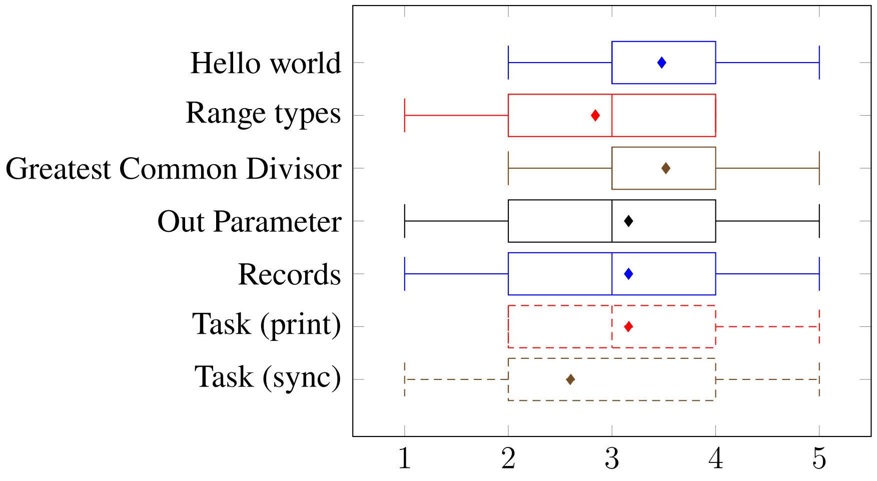

Figure 6.3: Results of General readability

Chapter 7

Conclusion & Future work

In order for organizations to be able to maintain their systems long term, it is highly important that they are modernized. Legacy systems are difficult and costly to maintain. However, migrations can be difficult and costly too. A source to source compiler helps in this regard, by automating part of the process.

In this thesis we approached the problem of modernizing Ada systems. We evaluated a variety of target languages and found Scala to be the most fitting target, due to its advanced type system, macro capabilities, implicit functions and constructors, and operator overloading.

We presented rules, a prototype implementing the rules, and the necessary runtime support libraries that allow modernizing to Scala the following Ada features:

- Ranged and fixed-point types
- Records
- Tasking system
- In and Out parameters
- Arrays
- Generics

Using only these features we were able to describe a variety of programs, which can now be automatically migrated. We, therefore, consider that we have fulfilled the goal of converting Ada code using advanced features to Scala.

Although these rules cover a sizeable subset of the Ada language, there is still room for further work on supporting more language features, for example:

- More synchronization primitives ("protected" types)
- Pre and post conditions (into Scala assertions)

- Real-time features (“delay” and “timed” statements)
- More standard library support

Finally, although this work focuses on translation between specifically Ada and Scala, there are general conclusions that can be drawn from this work. Particularly, Scala is a suitable target language for code translation that is readability-aware. Implicit functions allow the translated code to hide type and value transformations that would prove burdensome on translated code. Macro support allows code generation to occur during compilation, therefore reducing the amount of boilerplate shown to the programmer. Lastly, operator overloading allows custom operators to be defined, further improving the readability of the code.

Bibliography

[1] *1999 International Conference on Software Maintenance, ICSM 1999, Oxford, England, UK, August 30 - September 3, 1999*. IEEE Computer Society, 1999.

[2] *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004.

[3] AdaCore. libadalang. `https://github.com/AdaCore/libadalang`, Dec 2019.

[4] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In de Boer et al. [13], pages 364–387.

[5] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In Barthe et al. [6], pages 49–69.

[6] Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers*, volume 3362 of *Lecture Notes in Computer Science*. Springer, 2005.

[7] Keith H. Bennett. Legacy systems: Coping with success. *IEEE Softw.*, 12(1):19–23, 1995.

[8] Judith Bishop and Antonio Vallecillo, editors. *Objects, Models, Components, Patterns - 49th International Conference, TOOLS 2011, Zurich, Switzerland, June 28-30, 2011. Proceedings*, volume 6705 of *Lecture Notes in Computer Science*. Springer, 2011.

[9] Giuseppe Castagna, editor. *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*. Springer, 2009.

[10] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In Johnson and Gabriel [16], pages 519–538.

[11] Edmund M. Clarke and Andrei Voronkov, editors. *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*. Springer, 2010.

[12] Brad J. Cox and Kurt J. Schmucker. Producer: A tool for translating smalltalk-80 to objective-c. In Meyrowitz [22], pages 423–429.

[13] Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors. *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*. Springer, 2006.

[14] Digital Research, Inc. *XLT86 8080 to 8086 Assembly Language Translator USER'S GUIDE*, 1981.

[15] Stuart I. Feldman, David M. Gay, Mark W. Maimone, and N. L. Schryer. A fortran-to-c converter. *Computing science technical report*, 149, 1990.

[16] Ralph E. Johnson and Richard P. Gabriel, editors. *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. ACM, 2005.

[17] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA* [2], pages 75–88.

[18] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Clarke and Voronkov [11], pages 348–370.

[19] K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In Castagna [9], pages 378–393.

[20] UniqueSoft LLC. Performance of java code translated from cobol.

[21] Andrew J Malton. The software migration barbell. In *ASERC Workshop on Software Architecture*, 2001.

[22] Norman K. Meyrowitz, editor. *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), Orlando, Florida, USA, October 4-8, 1987, Proceedings*. ACM, 1987.

[23] Alok Srivastava and Jeff Boleng, editors. *Proceedings of the 2010 Annual ACM SIGAda International Conference on Ada, Fairfax, Virginia, USA, October 24-28, 2010*. ACM, 2010.

[24] Ricky E. Sward. The rise, fall and persistence of ada. In Srivastava and Boleng [23], pages 71–74.

[25] Mikio Takeuchi, Yuki Makino, Kiyokuni Kawachiya, Hiroshi Horii, Toyotaro Suzumura, Toshio Suganuma, and Tamiya Onodera. Compiling x10 to java. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop on - X10 '11*, pages 1–10. ACM Press.

[26] Andrey A. Terekhov and Chris Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, 2000.

[27] Marco Trudel, Carlo A. Furia, Martin Nordio, Bertrand Meyer, and Manuel Oriol. Automatic translation of C source code to eiffel. *CoRR*, abs/1206.5648, 2012.

[28] Marco Trudel, Manuel Oriol, Carlo A. Furia, and Martin Nordio. Automated translation of java source code to eiffel. In Bishop and Vallecillo [8], pages 20–35.

[29] Martin P. Ward. Assembler to C migration using the fermat transformation system. In *1999 International Conference on Software Maintenance, ICSM 1999, Oxford, England, UK, August 30 - September 3, 1999* [1], pages 67–76.

[30] Richard C. Waters. Program translation via abstraction and reimplementation. *IEEE Trans. Software Eng.*, 14(8):1207–1228, 1988.

[31] Felix Winkelmann. CHICKEN scheme - a practical and portable scheme system.

[32] Kazuki Yasumatsu and Norihisa Doi. Spice: A system for translating smalltalk programs into a C environment. *IEEE Trans. Software Eng.*, 21(11):902–912, 1995.