Universidade do Minho
Escola de Engenharia

Eduardo Augusto Peixoto da Silva Brito

A Formal Approach for a Subset of the SPARK Programming Language Semantics and Program Verification

Outubro de 2010

Universidade do Minho
Escola de Engenharia

Eduardo Augusto Peixoto da Silva Brito

A Formal Approach for a Subset of the SPARK Programming Language Semantics and Program Verification

Dissertação de Mestrado
Mestrado em Informática

Trabalho efectuado sob a orientação do
Doutor Jorge Sousa Pinto
e co-orientador do
Doutor Luís Pinto

Outubro de 2010

Acknowledgements

I would like to thank all the people that were involved in the making of this thesis, in one way or the other.

My advisors, for all the help and guidance they provided and for going "beyond the call of duty" several times. I would also like to thank Yannick Moy, Rod Chapman and Tristan Crolard for helping me with related work being done in the SPARK community.

My friends for their support and for preventing me from working myself to death (mostly by helping me procrastinate with videos of fluffy kittens and by being awesome). A special thanks is also in order to the people in Braga's Kendo club that, besides being my friends, have made the training so fun and helped me relax from the frustrations of work.

Last but definitely not least, my family for supporting me and my bad(er) temper, induced by thesis-related stress and sleep deprivation.

Finally, FCT for funding the research. This work was partially funded by a research grant of the FCT-funded RESCUE project (PTDC/EIA/65862/2006).

Resumo

SPARK é uma linguagem de programação e um conjunto de ferramentas que é usado para o desenvolvimento de software para sistemas safety-critical. SPARK tem vindo a receber a atenção do meio industrial e académico devido ao sucesso que tem tido em aplicar métodos formais a casos industriais reais. Acreditamos que a linguagem de programação SPARK possa ser usada como uma plataforma de trabalho para desenvolver de forma aprofundada teorias e ferramentas para verificação de programas, incluindo concorrência, tempo-real e o desafio do compilador verificado, assim como outros desafios de verificação, e que estes estudos possam ter benifícios para ambos os campos.

Nesta tese escolhemos um suconjunto sequential da linguagem de programação SPARK e formalizamos a sua semântica. Durante a nossa investigação, implementamos um interpretador para a nossa linguagem de programação e propomos trabalho futuro e ferramentas que podem ser desenvolvidas no contexto deste *framework*.

Palavras-chaves: Semântica Formal; Verificação de Programas; SPARK.

Abstract

SPARK is a programming language and toolset, used for the development of software for safety-critical systems. SPARK has garnered attention from both the industrial and academic arenas due to its success in applying formal methods to real industrial cases. We believe that the SPARK programming language can be used as a workbench to further develop theories and tools about program verification, including concurrency, real-time and the verified compiler grand challenge, as well as other verification challenges, and that these studies may benefit both arenas.

In this thesis we have chosen a sequential subset of the SPARK programming language and formalized its semantics. We have also implemented an interpreter for our programming language during our research and we propose future work and tools that can be done in the context of this framework.

Keywords: Formal Semantics; Program Verification; SPARK.

Contents

List of Acronyms

ATP	Automated Theorem Provers
BISL	Behavioral Interface Specification Language
DbC	Design by Contract^{TM}
DDC	Dansk Datamatik Center
EBNF	Extended Backus-Naur Form
EPL	Embedded Programming Language
FM	Formal Methods
FOL	First Order Logic
HOL	Higher Order Logic
NS	Natural Semantics
PCC	Proof Carrying Code
PO	Proof Obligation
RAC	Runtime Assertion Checking
SL	Specification Language
SMT	Satisfiability Modulo Theories
SOS	Structural Operational Semantics
SPARK	Spade Ada (?) Kernel
VC	Verification Condition
VCGen	Verification Condition Generator

List of Figures

List of Tables

Type Signatures

Inference Rules

Definitions

Proof cases

Code Listings

1 Introduction

The general scope of this thesis is the development of safety-critical software; more precisely, software written in the Ada programming language. Guidelines and standards have been encouraging the use of more rigorous software practices, including formal methods. The SPARK programming language, a subset of Ada and a toolset, is used to perform program verification at the industrial scale, and is widely used to promote such rigorous practices.

In this thesis we present our work on the formalization of a subset of SPARK, and our research related to the language and its use in the context of formal methods. Let us briefly look at each of the above issues in turn.

The Ada programming language Ada, since its inception, has been focused on providing features to make programming safer. The Ada programming language has gone through several official revisions (Ada 83, Ada 95, Ada 2005) and is now undergoing a new revision, that will be named Ada 2012. These revisions to the language have kept it updated, regarding developments in the programming languages community, and have fulfilled the needs of people using the language for software development in the safety-critical arena. Ada has also a tradition of being used in education, as part of the computer science curricula of several universities, especially in the United States of America and in some countries of Europe such as Spain, France and the United Kingdom.

Safety-critical systems Safety-critical systems demand that software meets the highest standards of quality and correctness. This demands that rigorous approaches to software development are used, in all areas of the software development process.

Ada is one of the most used programming languages in the area of High Integrity Software/ Safety-critical systems. In certain domains, such as defence, aeronautics and aerospace, Ada is considered to be the de facto standard.

Guidelines and standards Guidelines for software development have addressed ambiguity problems found in programming languages, good software practices and restrictions to programming languages, that help in the verification, validation and certification process of software. Some guidelines and standards, such as DO-178C and Common Criteria, have also began to officially recognize formal methods as a helpful tool to achieve more guarantees of correctness and a more rigorous software development practice.

The SPARK programming language In the confluence of these things we can find the SPARK programming language. The SPARK programming language is a restricted subset of the Ada programming language, with added annotations for dataflow and program verification. SPARK also provides its own toolset, with tools such as a static analyzer, a verification condition generator, an automated theorem prover and an interactive prover, that enforces the restrictions to the language, and also allows to prove properties of the source code up for analysis.

Our work In this thesis we present work on a formal approach for a subset of the SPARK programming language. While SPARK provides a methodology for program verification, some of its theoretical bases are not specified anywhere, such as its program logic and verification condition generator. SPARK has been a very interesting language to work with; although it is relatively small compared to full Ada, it is useful for the safety-critical domain and it presents a wide range of challenges in its formalization. These challenges have been very educational and have shown us the effort and technical details that we need to address when considering a real programming language. This work has also opened up many opportunities for future research in the intersection between safety-critical software and formal methods.

Contributions Section 2.2 was accepted as a regular paper at the 15th International Conference on Reliable Software Technologies - Ada-Europe 2010. This paper was a comparative case study between SPARK and Frama-C/ACSL. While both languages and tools have several differences, Ada (which SPARK is a subset of) and C are widely used in the development of software for safety-critical systems and the paper presented a comparison of what it is possible to do regarding formal verification on SPARK and C. The paper is not reproduced here in its entirety but has been adapted to fit the structure of the thesis.

Sections 1.4, 1.5 and 1.6 are from an accepted paper at the EVTSIC'10 track at INForum 2010 called "A (Very) Short Introduction to SPARK: Language, Toolset, Projects, Formal Methods & Certification". In this paper we presented a survey regarding SPARK, with a focus on what kind of features SPARK has and where it is being used and how. This paper has also been posted at the resources webpage of the Hi-Lite project.

Chapters 3 and 4 are the core of the thesis (although it has not been published yet). The contributions that these chapters brings are the design of a subset of SPARK, called mSPARK, the definition of its informal static semantics and natural semantics, as well as the definition of a program logic for mSPARK, with safety restrictions and range types, and the proof of soundness for this program logic. We believe this to be a valuable contribution since this was missing from SPARK.

Communications At the beginning of the thesis we presented a fast abstract at the European Dependable Computing Conference 2010 (EDCC-8), describing statement of purpose of the thesis. This fast abstract is called “Enhancing Program Verification for SPARK”. At the same conference we also presented a poster entitled “A Formal Approach for the mSPARK Programming Language” with specific references to the work on mSPARK that we were developing.

Awards A free registration for the QUATIC 2010 conference was awarded to the author of the thesis after entering a contest with a text describing this work.

Organization The remainder of the introduction gives light introductions to several topics. These topics and their underlying foundations motivate and are used in our own research work, that is presented here in the thesis. In Chapter 2 we present related work to program verification in SPARK and we also note some previous approaches at the formalization of Ada and SPARK. Chapter 3 presents our subset of the SPARK language, mSPARK, ranging from the abstract syntax to its formal operational semantics. Chapter 4 presents the program logic we have developed for mSPARK and the Verification Condition Generator specification and we provide soundness proofs for the program logic. Chapter 5 is the conclusion of the thesis an presents future work that can be done in mSPARK and SPARK.

The links to the publications and communications are available on my homepage[1].

1.1 Logic & Computer Science

Mathematical logic has been a prolific area in mathematics throughout the XX century, and now, in the XXI century. While most logics have been developed because mathematicians needed more powerful mechanisms to describe their theories, other areas of human knowledge have adopted the work that has been done in mathematical logic into their own work.

For computer science and computer scientists, some logics have proven to be quite helpful and some were even developed and/or changed/evolved to specifically address problems in the computer science domain.

First order and higher order logics have been widely used in the development of frameworks for specifying, understanding and proving properties of programs. While First Order Logic (FOL) is usually used in program specifications and source code annotations, Higher Order Logic (HOL) is used, for instance, by interactive provers such as Isabelle/HOL to discharge verification conditions that were not discharged by automated theorem provers. The need for interactive provers/proof assistants arises from the fact that any logic with some expressive power (e.g. FOL) can not have a decidable algorithm that, for any statement of the logic, is able to prove or disprove it.

[1]http://alfa.di.uminho.pt/~edbrito

Temporal logics have also been used in computer science for many useful things. It is an essential formalism that underlines several model checkers, especially model checkers dealing with concurrency and real-time behaviour. Being the basis for such tools and given the nature of these tools, it has been widely used in several settings, from academia to industry, from software to hardware.

Algebraic logic and modal logic (which includes dynamic logic) have also found uses in computer science. Algebraic logic, for instance, is part of CafeOBJ for algebraic specification of systems and dynamic logic is part of the KIV tool and of the KeY tool.

Detailed information on logic & computer science can be found in [VR01, Hed04, BA93].

1.2 Semantics of Programming Languages

This section briefly introduces the notion of semantics of programming languages. Semantics are what defines the meaning of programming language constructs.

We present several notions related to programming language semantics, ranging from abstract notions such as static and dynamic semantics, which can be applied to a wider context, to notions of formal semantics, such as structural operational semantics and natural semantics, which have a very specific meaning.

Ours is a brief overview of these topics. We focus only on the notions of semantics that we need for our work, presenting only a brief overview of other possible semantics frameworks. A deeper treatment of these topics can be found in [Plo81, Kah87, Win93, Hen90, NN07, Rey99].

Operational Semantics We describe now two abstract notions of operational semantics: static and dynamic semantics.

Static Semantics This refers to the meaning that language constructs have statically, e.g. at compile time. We can have a syntactically valid statement such as "x := 10;" but this may be semantically invalid for several reasons. One such reason is that the variable may not be in scope or that x is not of a numeric type. Note that this depends on the definition given by the static semantics and not the syntax; for a programming language that does not need variable declarations, the previous statement would be perfectly valid. Static semantics reasons about the constructs of the language and defines what is and what is not valid for all classes of statements that the programming language may have.

Static semantics is restricted to things that can be asserted at compile time. Some languages, like Haskell [Jon03], can infer the types of functions and identifiers at compile-time; this can be part of the definition of a static semantics, where we add the inference algorithm to the specification. Other type systems, like Ada's type system, can not infer nor check completely the types of variables. In Ada this happens because types are limited/constrained by ranges and only at runtime, given certain values, a expression may evaluate to values that do not belong to that type. For this class of problems we need to have a dynamic semantics specification to tell how the language should proceed.

Note however that although we mentioned specifications and algorithms, static semantics is a general term that does not have a specific formalism to express its definitions. As an example, the Ada Reference Manual [TDB+07] (ARM) defines most of these properties in natural language.

Dynamic Semantics This type of semantics expresses what should happen when a language construct is executed in a given program state for a given environment. State refers to the values that objects in the program have, while environment is the collection of static information, such as types and declarations.

As an example, the statement "'x := y*z;" may be valid for our static semantics, that is, the types of the variables are in accordance and the variables are in scope but the result of the multiplication may be a value outside the range of x. In Ada this example would raise an exception with the name "Constraint_Exception". In languages such as C, x would originate overflow (or underflow) and no error would be generated.

While the static semantics would reject certain programs because they are not valid, with dynamic semantics, because the program is already executing, we need to specify what happens if no error occurs, what happens when an error occurs and if we can deal with it in any way, and most importantly, what is an error.

As with static semantics, the dynamic semantics concept does not adhere to any specific formalism and may be expressed in natural language.

Formal Operational Semantics After seeing abstract notions of operational semantics, we now present two well known formal operational semantics styles: Structural Operational Semantics and Natural Semantics.

Structural Operational Semantics It was first devised by Plotkin [Plo81] and is a very influential style of formal semantics. Structural Operational Semantics (SOS) defines rules for programming languages by using transition systems.

An important point of SOS is that it was developed so that it would be possible to describe the computation for language constructs in very small steps (small-step semantics is another name used to refer to this style of semantics). While this may seem unnecessary for most constructs, where we only want the final result of a computation, certain things like non-determinism and concurrency can easily be described in a semantics system like SOS.

Natural Semantics While in SOS we are concerned in describing fine grained computations of programming language constructs, Natural Semantics (NS) abstracts these details. In NS we are more interested in formally describing a programming language at a higher level, where we take large steps in computations, that is, we do not care for possible intermediate computations that a certain construct may need to do to properly execute (this is why this style of semantics is also called big-step semantics, as a contrast to small-step semantics).

We are interested in the execution order of program constructs and in their “inputs” and “outputs”, as a black box of sorts. This allows for a higher level of reasoning when thinking of the semantics of the programming language. This comes at the cost that certain things are not possible (or at least, they are not immediate) to be expressed in this style, such as concurrency.

A side effect of this big step evaluation style of semantics is that we can argue that this style describes an abstract machine and/or interpreter for a given language. As we shall see, we take advantage of this property in the work that we developed.

It should also be noted that although we are separating SOS from NS some authors argue that they are the same formalism and in fact, we can prove the equality between NS and SOS specifications. Examples are given in [NN07].

Axiomatic Semantics Axiomatic Semantics is a way to specify the logical meaning of the programming language constructs. This allows us to infer and/or specify certain properties about the source code.

Although operational semantics can be used to verify properties of the programs through execution, axiomatic semantics is meant to assert properties based on the logic of the constructs in a static way. It should be noted that axiomatic semantics is not meant to replace operational semantics. In fact, both semantics should be used together. While the operational semantics defines how we should execute the constructs, the axiomatic semantics determines the logic meanings associated with the constructs. To verify that our axiomatic system is sound, we need to check if the rules we defined for it are in accordance to what is expected from operational semantics. In other words, the axiomatic semantics needs an operational semantics so that it can be shown that it is sound, but we also need the axiomatic semantics to reason about the meaning of our programs and their properties, before we operationally execute them.

Other Styles of Semantics We have presented the most common and used styles of semantics. Most of these styles are given at undergraduate level courses. Some other styles of semantics that are used are Modular SOS, created with the purpose of lessen the impact of changes in SOS specifications, Action Semantics and Denotational Semantics. Denotational Semantics is also an widely studied and used style of formal semantics, usually studied at the same time as operational and axiomatic semantics, although we did not addressed it here.

1.3 Program Verification

Floyd-Hoare Logic Robert W. Floyd proposed in 1967 [Flo67] the use of flowcharts for giving formal definitions to the meanings of programs so that proofs of correctness, equivalence and termination could be given. Tony Hoare in 1969 [Hoa69] developed the mathematical framework, that later would be known as Hoare Logic, which allowed for logical reasoning about computer programs. He also proposed a development methodology where the programmer would write what was expected before executing a command and what should happen after its execution. This notion is represented in what is known as Hoare triples. Hoare triples come in the form of {P} C {Q}, where C is the command to be executed, P and Q are assertions, where P is the precondition to the execution of command C and Q is the postcondition which should be true after the execution of the command. The notation used in the presented Hoare triple is for what is called "partial correction", where nothing is stated about the termination (or non-termination) of the program. Total correctness in Hoare logic is represented by [P] C [Q]. The notion of total correctness includes the termination of command C in the Hoare triple.

Weakest Preconditions Dijkstra would later develop the notion of predicate transformers [Dij75]. This led to what is now known as Dijkstra's weakest precondition calculus. A weakest precondition is a logical statement that, given an Hoare triple {P} C {Q}, is the weakest P which ensures Q.

This is very useful for program verification. Given this calculus and a Hoare Logic, we can construct most assertions automatically, especially for (usually) trivial cases such as assignment. Refinement, as in B's abstract machine notation (AMN) [Abr96], also uses this notion of predicate transformers.

Verification Conditions & Generators In the traditional Hoare Logic framework there usually exists a consequence rule. This rule is very helpful for doing manual proofs in the framework of Hoare Logic. The problem is that this rule introduces ambiguity in our system and it makes the automatic generation of verification conditions (VC) difficult.

VCs, also known as Proof Obligations (PO), are properties that we want to prove/ensure about our programs. A VC generator (VCGen) is an algorithm which generates verification conditions, normally using an Hoare-like logic and Weakest Precondition Calculus.

One approach that is taken to automate this process is to remove the ambiguity introduce with the consequence rule by rewriting the program logic into a goal-oriented program logic that does not depend on the consequence rule.

Theorem Provers After having VCs, we want to prove that our program is correct by proving (usually called discharging) the VCs that have been generated by a VCGen. Proofs can be done with pen and paper or we can use theorem provers. These theorem provers can be automated theorem provers (ATP) or interactive (also called proof assistants).

While theorem provers may be general purpose and not only used for program verification, there has been an investment on the development of satisfiability modulo theories (SMT) provers, which already provide some theories aimed specifically at program verification.

Other Forms of Program Verification and Rigorous Software Development So far, we focused on program verification based on source code, VCs and correctness proofs. We have excluded other formal approaches such as model checking and program construction.

Model checking may use arbitrary models or models constructed from source code (software model checkers) and may be used to check properties over those models, which may be more or less abstract. Program construction is a correct-by-construction approach where we calculate the program from its specification. This calculation is done by refining the specification and each step of the refinement is proved to be correct, regarding the previous step. At the end of the refinement we obtain a program that is shown to be correct by construction regarding its initial specification since every step of the refinement is proven to be correct.

We also overlooked other predicate transformers such as strongest postcondition. In this calculus, instead of providing the weakest precondition that satisfies the postcondition for the Hoare triple, we calculate the strongest postcondition that the execution of the command must ensure.

Further information on these topics can be found at [AFPMDS10].

1.4 Contracts & Specification

In this section we present Design-by-Contract (DbC) and in what ways it resembles and differs from Behavioral Interface Specification Languages (BISL). It is important to describe this since the language SPARK which we will be focusing on (as well as several others that aim at source code verification) has its approach rooted in this.

Design by Contract™ The term "Design by Contract" was first coined by Bertrand Meyer [Mey92] and is largely associated with the Eiffel [Mey00] programming language, an Object-Oriented programming (OOP) language, as part of the language's philosophy and design process. The term is also a registered trademark and some authors prefer to use the expression Programming by Contract.

The main ideas behind DbC are: a) to document the interface of modules[2] and its expected behaviour, b) to help with testing and debugging and c) to assign blame when a contract is breached. This is achieved by having structured assertions such as invariants and pre- and post-conditions.

[2]The term modules is equivalent to package or class.

In DbC, following the tradition of Eiffel, assertions are boolean expressions, often using subprograms[3] written in the same language as the host programming language and are intended to be checked at runtime (*runtime assertion checking (RAC)*), by executing them. Writing assertions in this way is friendlier to developers but it makes formal verification difficult because contracts are also code and not mathematical specifications describing the properties to be ensured.

Recent approaches, such as Code Contracts [Cor10], uses abstract interpretation [FL10] to do semi-formal verification. Abstract interpretation is capable of finding some errors at the expense of not being completely precise and signalling false positives. We do not consider this as rigorous formal verification because of its lack of precision.

In [JM97] it was illustrated how Eiffel could have helped prevent the bug in the software of Arianne V, thus avoiding one of the most expensive software errors ever documented. What is stated in [JM97] is that the error that made Arianne V go wrong could have been avoided if the pre-conditions for the subprogram that failed had been clearly stated in the code. If the dependencies were clearly documented in the code then the verification & validation team would have been aware of what could (and did) generate a runtime error.

To sum it up, DbC is used to document source code and to have the program checked while it is executing, using structured annotations that are written as boolean expressions of the host programming language.

Behavioral Interface Specification Languages Behavioral Interface Specification Languages (BISL) was a term introduced with Larch[GHG+93]. The Larch family of languages had two-tiered specifications; one language addressed general properties of components and communication between them, the LSL (Larch Specification Language), and a family of languages addressed particularities of each target language (Larch supported C, C++, Smalltalk, Modula-3 and Ada). These specifications were written in a mathematical notation and could be related directly to the source code, using the interface language for each of the target language. This differs greatly from traditional specification languages (SL), such as Z[Spi89] and VDM[Jon90], where the specification is written as a separate entity with no (direct) relation to an existing implementation.

Larch was focused mainly on specifying, with brevity and clarity, the interface of program components. Although some specifications were executable, executability of specifications was not an objective of the Larch family of languages; this is the exact opposite of DbC. The way that specifications are written in BISLs and SLs are similar. In both approaches, the specifications are written using a well-defined formal notation that is related to a well-defined formal logic. These formal definitions do not use expressions from the host language although they may look similar in some cases; this is another difference regarding DbC.

[3]Subprograms is being used as a more generic way to refer to methods, functions, procedures and subroutines.

It may also be possible to animate/execute specifications written in some SLs (depending on the available tools) but this is different from animating/executing DbC and BISL specifications because we are animating a logical specification, without a corresponding implementation, and not an specification associated to an implementation.

While writing annotations in a mathematical notation is very expressive and particularly helpful for program verification, excessive mathematical notation can lead to the poor adoption of a BISL. JML[LBR99] is a modern example of a BISL, rooted on the principles of Larch, which has taken this into account. JML avoids excessive mathematical notation, while having a mathematical background, and has gained supporters in the academic and industrial arena, especially with the success of JavaCard[Sun00] verification tools[AR02, MMU04].

JML, as noted in[LCC$^+$05, BCC$^+$03], is associated with a set of tools that makes possible to overcome the typical non-executable nature of BISLs. Also, besides being able to do RAC, it also allows for the formal verification of Java programs, given the right tools/frameworks. ACSL[4][BFM$^+$09a] is another interesting and modern BISL. While it has a large influence from JML, it has greater expressive power regarding the definition of mathematical entities.

There is an excellent and extensive survey[HLL$^+$09] on this subject, focusing on automatic program verification and the Verified Software Initiative[HM08].

1.5 SPARK

In this section we describe the features of the SPARK programming language and how it is supposed to work with the help of its toolset. We will focus mainly on program verification and proof support for SPARK, with the tools that are available from the official toolset distributed by AdaCore and Altran Praxis. This section assumes SPARK GPL 2009 (version 8.1.x), unless it is stated otherwise.

The Programming Language SPARK stands for SPADE Ada Kernel[5]. SPADE was a previous project from Program Verification Limited, with the same aims as SPARK, but using the Pascal programming language. The company later became Praxis High Integrity Systems and it is now called Altran Praxis.

SPARK is both a strict subset of the Ada language, augmented with source code annotations, and also a toolset that supports its methodology. The SPARK annotations can be thought of as a BISL.

It is a strict/true subset of the Ada language because every valid SPARK program is a valid Ada program; in fact, there is no SPARK compiler. SPARK depends on Ada compilers to generate binary code. SPARK was also cleverly engineered so that the semantics of SPARK programs do not depend on decisions made by a specific compiler implementation

[4]ANSI/ISO C Specification Language.

[5]The R in SPARK is only there for aesthetic purposes.

(e.g. whether a compiler chooses to pass parameters of subprograms by value or by reference). Although outdated, SPARK also has a formal semantics [O'N94] that is defined using a mixture of operational semantics and Z notation.

SPARK removes some features of the Ada language such as recursion, dynamic memory allocation, access types[6], dynamic dispatching and generics. It also imposes certain restrictions on the constructs of the language (e.g. array dimensions are always defined by previously declared type(s) or subtype(s)[7]).

On top of the language restrictions, it adds annotations that allow for the specification of data-flow, creation of abstract functions and abstract datatypes and to the use of structured assertions (loop invariants are available but package invariants are not). There is also a clear distinction between procedures (which may have side-effects) and functions (which are pure/free of side-effects and have to return a value).

Ada has the notions of package (specification) and package body (implementation); these are the building blocks for Object-Oriented Programming (OOP) in SPARK and Ada, although we may choose not to use the OOP features of the language. Even without using OOP we can still use notions such as encapsulation. SPARK restricts OOP features that make the code hard to verify, such as dynamic dispatching. This issue is also addressed in [Bar08], related to Ada 2005.

Package specifications can have abstract datatypes and abstract functions, which can then be used to define an abstract state machine. When coding the implementation, the abstract datatypes have to be refined into concrete datatypes, using the *own* annotation, and the definitions for abstract functions are written into *proof rule* files. These files are used when trying to discharge verification conditions. By providing this additional layer of abstraction, we can reason about packages as components where we know how they behave without depending on a given implementation.

Although it is not our aim to talk about the information flow capabilities of SPARK in this thesis, it is interesting to note that in SPARK Pro 9, which has the SPARK 2005 version of the language, it is also possible to validate secure and safe flow of information between different safety integrity levels and information security levels (e.g. unclassified, top secret).

All these features enable the possibility of developing software using SPARK's Correctness by Construction approach where a package is designed and developed with the aim of being formally verified and to use and be used by other formally verified packages.

Finally, it should be noted that SPARK has support for a subset of Ravenscar[8] dubbed RavenSPARK [SPA08].

[6]Users not familiar with Ada should note that these are equivalent to pointers.

[7]In Ada, the type mechanism allows for the definition of *ranges*. These ranges are limited by a lower and upper bound, known as T'First and T'Last where T is the type. It is also possible to get the range using T'Range.

[8]Ravenscar is a limited subset of concurrency and real-time of the Ada language.

Toolset & Program Verification The SPARK toolset is what enables the use of SPARK. By not having a compiler, it must have a tool that checks the restrictions of the SPARK language; this tool is called the Examiner. The Examiner is also the verification condition (VC) generator (VCGen).

The proof tools of the SPARK toolset can be used to discharge the VCs. The Simplifier is the automated theorem prover and tries to discharge the VCs by using predicate inference and rewriting. While the tool is very successful in discharging VCs related to safety [JES07], it is not as capable of discharging VCs related to functional correctness as other modern provers [JP09]. The Proof Checker is the interactive prover/proof assistant of the toolset.

An auxiliary tool, that ties all these tools together, is called POGS. POGS stands for Proof ObliGation Summarizer. It generates a structured report with much information, including warnings and errors related to the code but, most importantly, it constructs the report regarding the VCs that have been proven valid; those that remain to be proven; and those that have been shown to be invalid.

In Fig. 1.1 we illustrate the typical usage of the SPARK toolset. It should be noted that dashed arrows represent optional paths in the graph. POGS is not "dashed" because it is helpful in describing what remains to be done and is thus more often used than not.

The SPARK Pro 9 toolset also has another tool called the ZombieScope. This tool provides analysis related to dead paths. A dead path is a piece of code that will never be reached; this is most likely caused by programming errors. This tool generates dead path conditions (similar to VCs) and tries to disprove that something is not reachable; if it fails, then it has found a dead path. This information also appears on the report generated by POGS, as expected.

The GPS (GNAT Programming Studio), which is provided by AdaCore, has support for SPARK and it has been constantly updated to make it easier to use the SPARK toolset inside an IDE[9] that is familiar to some Ada users. GPS is not integrated in the toolset but it is tightly coupled to it.

It should also be noted that the SPARK toolset does not provide any means to do RAC.

[9]Integrated Development Environment.

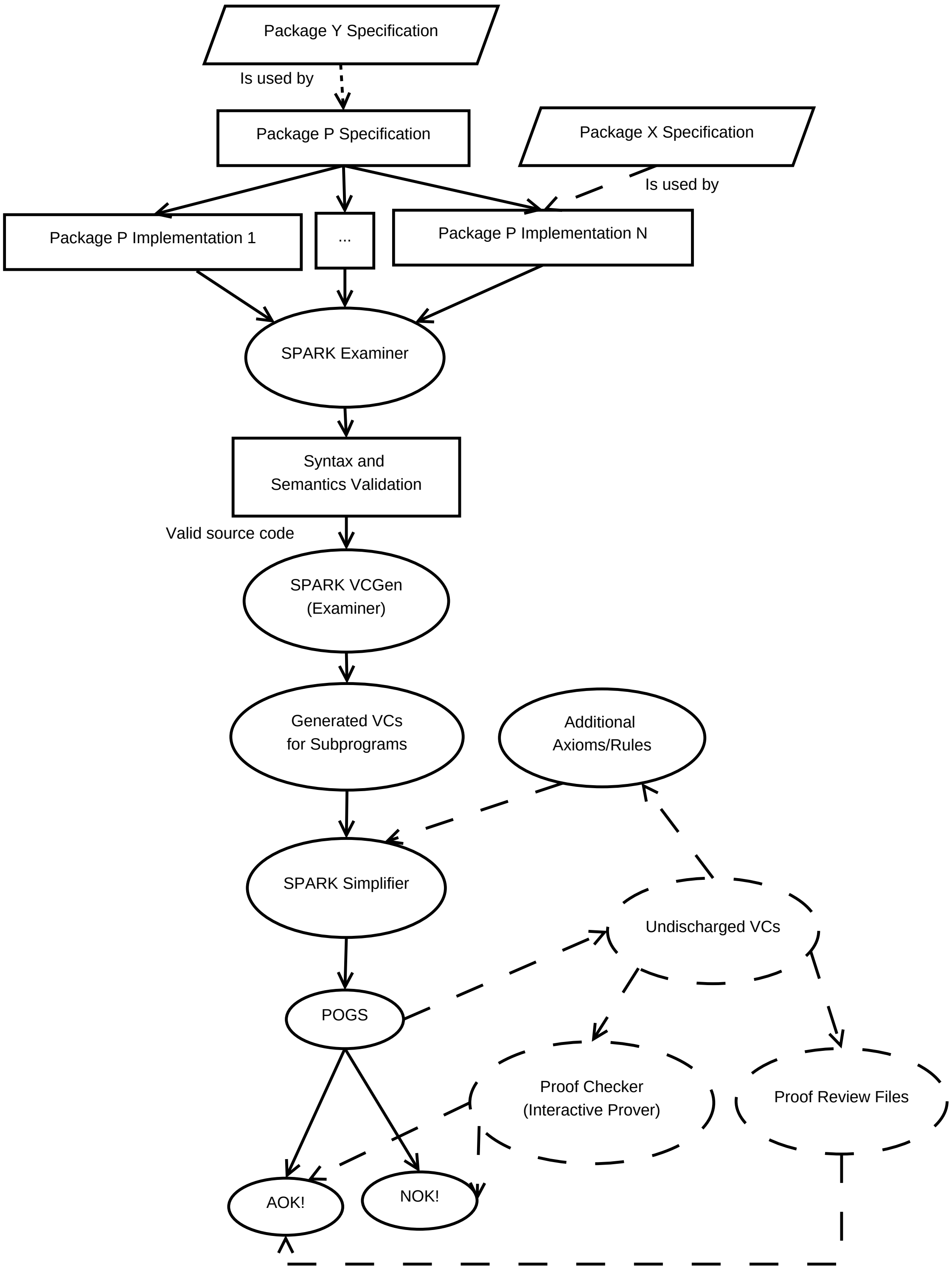


Figure 1.1: Using the SPARK toolset to preform formal verification.

1.6 Projects & Methodologies Related to SPARK

In this section we illustrate the capabilities of SPARK by considering both projects where the language has been used and also some methodologies and tools that either use or generate SPARK code. As a preview, we will talk about Tokeneer, Echo, SCADE and the C130J helicopter in this section.

Industrial & Academic Projects Tokeneer [BCJ+06] was an industrial project developed under a contract with the NSA, to show that it was possible to achieve Common Criteria EAL5 in a cost effective manner. The purpose of the project was to develop the "core" part of the Tokeneer ID Station (TIS). The TIS uses biometric information to restrict/allow physical access to secured regions.

The project was successful, exceeding EAL5 requirements in several areas, and was allowed to be publicly released during 2009, along with all deliverables and a tutorial called "Tokeneer Discovery". In 9939 lines of code (LOC), there were 2 defects; one was found by Rod Chapman, using formal analysis, and another during independent testing, thus achieving 0,2 errors per kLOC. Tokeneer has been proposed as a Grand Challenge of the Verified Software Initiative [Woo].

Praxis also published a short paper [Cha00] where their industrial experience with SPARK (up until 2000) was described. SHOLIS (a ship-borne computer system), MULTOS CA (an operating system for smart-cards) and the Lockheed C130J (Hercules) helicopter, for which SPARK was used to develop a large part (about 80%) of the software for the Mission Computer, are all briefly presented in this paper.

These case studies illustrate several features of the SPARK language, including how the language eases MC/DC[10] verification by having simpler code, how proofs are maintained and can be used as "regression proofs" instead of "regression testing" and how SPARK can also inhabit multi-language systems. They show that SPARK is not appropriate to being adopted late in the development, especially when trying to re-engineer Ada code into SPARK.

Recently [Alt] SPARK has been chosen as the programming language for a new NASA project, a lunar lander mission. SPARK will be used to develop the software of the CubeSat project. SPARK was also used for a study on verified component-based software, based on a case study for a missile guidance system [LW07]. This goes hand-to-hand with the SPARK philosophy of Correctness by Construction.

The Open-DO initiative[11] is also developing a project, called Hi-Lite[12], which uses SPARK in several ways. One of the aims of the project is to create a SPARK profile for the Ada language[13]. While SPARK itself can be viewed as a profile for Ada, the aim of Hi-Lite is

[10]Modified Condition Decision Coverage.
[11]http://www.open-do.org
[12]It should be noted that this project is very large and this is just a small portion of the aims of that project.
[13]This is being called sparkify.

to provide a less restrictive SPARK, with the benefits that SPARK has shown over time. Another aim of the project is to write a translator from SPARK (or sparkify) to Why [Fil], so that it is possible to use Why's multi-prover VCGen. Why's VCGen has support for interactive provers as well as automated provers. This would provide an alternative toolchain for SPARK besides the one that is maintained by Altran Praxis.

There is also ongoing research work by Paul B. Jackson [JP09] in translating the VCs generated by the Examiner to SMT-Lib input. This work, that originated from the academia, is scheduled to be included in the SPARK Pro toolset as an experimental feature, in a future release.

Industrial Tools & Methodologies The SCADE Suite is a well known software design tool that has been used, for example, by Airbus in the development of the Airbus A340 and is being used in several of the Airbus A380 projects. The SCADE Suite has a certified code generator for SPARK [AD06]. Perfect Developer [Cro03] is another tool for modelling software for safety-critical systems that has a code generator for SPARK. It is out of the scope of this thesis to try to thoroughly explain these tools but, in a very simple way, these tools are driven by model engineering in a formal setting and have been used, especially SCADE, in large scale projects.

SPARK is also being used as an essential component of the Echo approach. This approach [SYK05] is able to tightly couple a specification (in PVS [ORS92]) with an implementation. It generates SPARK annotations from the PVS specification; this generated code is then completed to form the implementation. Afterwards, the tool extracts properties from the implementation to check if the implementation conforms to the specification. This approach has been used to formally verify an optimized implementation of the Advanced Encryption Standard [YKNW08].

2 Program Verification in SPARK

After presenting the concepts and a small introduction to SPARK, we now present the work that has already been done in Ada and SPARK formalization and verification.

We also present a comparison with ACSL/Frama-C, which is a modern program verification framework for the C programming language. The reason for this comparison is that while C and SPARK are quite different, they are the most used programming languages in the development of safety-critical software. Also, ACSL/Frama-C is a state of the art tool for verification of C programs, as SPARK is for (a subset) of Ada.

This comparison is also used as a light tutorial and introduction to both tools.

2.1 Related Approaches to Ada and SPARK Formalization and Verification

There were several attempts to formalize Ada and subsets of Ada since the inception of the programming language.

In 1980, a book was published describing what steps should be taken so that there could exist a formal description of the static and dynamic semantics of Ada [BO80a]. The company that won the contract to develop the first Ada compiler, Dansk Datamatik Center (DDC), had the obligation to develop a formal semantics for the language [BO80b, CO84]. They tried to develop the semantics in the denotational style but eventually gave up on this effort.

Later in the 1980s, there were three significant attempts at formalizing Ada for program verification: Anna [LvHKBO87], Penelope [Ram89, Gua89, GMP90] and Ava [Smi95]. Anna added annotations to Ada code for program verification, Penelope was a verification framework that had an associated development methodology, using Larch/Ada, and Ava was a formalization of the semantics of a subset of Ada.

Around the same time, Program Verification Limited (a company), was developing SPARK, using tools and knowledge they had acquired from the development of SPADE. The documents describing the formal semantics of SPARK were completed and made available in 1994; these documents [O'N94] describe the SPARK83 version of the SPARK language.

At the time NASA was working on the development of a formal semantics for Ada 9X (which later would become Ada 95).

Most of these efforts were later discarded and left incomplete or unavailable. SPARK has been the only of those efforts that have resisted up until now, with an active academic community with publications on the language, approach and toolset, as well as publications on case studies where it has been used successfully.

Unfortunately, even SPARK has not kept its formal semantics specification up-to-date, being the 1994 document on SPARK83 the latest document that is available on the formal (static and dynamic) semantics. A major flaw that SPARK has is the lack of a formal specification for the program logic and the verification condition generator.

2.2 Program Verification in SPARK and ACSL: A Comparative Case Study

We[1] present a case-study of developing a simple software module using contracts, and rigorously verifying it for safety and functional correctness using two very different programming languages, that share the fact that both are extensively used in safety-critical development: SPARK and C/ACSL. This case-study, together with other investigations not detailed here, allows us to establish a comparison in terms of specification effort and degree of automation obtained with each toolset.

2.2.1 ACSL/Frama-C

Frama-C [BFM+09a] is a tool for the static analysis of `C` programs. It is based on the intermediate language Jessie [BFM+09b] and the multi-prover VCGen Why [FM07]. `C` programs are annotated using the ANSI-C Specification Language (ACSL). Frama-C contains the `gwhy` graphical front-end that allows to monitor individual verification conditions. This is particularly useful when combined with the possibility of exporting the conditions to various proof tools, which allows users to first try discharging conditions with one or more automatic provers, leaving the harder conditions to be studied with the help of an interactive proof assistant. For the examples in this paper we have used the `Simplify` [DNS05] and `Alt-Ergo` [CCK06] automatic theorem provers. Both Frama-C and ACSL are work in progress; we have used the Lithium release of Frama-C.

2.2.2 Running Example - Bounded Stack

Figure 2.1 shows a typical specification of a bounded stack that can be found in many tutorials on the design by contract approach to software development [Mey92]. The figure contains an informal description of each operation on stacks, and in some cases a contract consisting of a precondition and a postcondition.

Notice that methods `count`, `capacity`, and `isFull` occur in several preconditions and postconditions. In fact, the first two are not usually given as part of a stack's interface, and

[1]Jorge Sousa Pinto was the other author of this paper

- `nat count()` – Returns the number of elements currently in the stack.
- `nat capacity()` – Returns the maximum number of elements that the stack may contain.
- `boolean isEmpty()` – Returns information on whether the stack is empty.
  Postcond: `Result = (count() = 0)`
- `boolean isFull()` – Returns information on whether the stack is full.
  Postcond: `Result = (count() = capacity())`
- `int top()` – Returns the top of the stack.
  Precond: `not isEmpty()`
- `void pop()` – Removes the top of the stack.
  Precond: `not isEmpty()`; Postcond: `count() = old_count() - 1`
- `void push(int n)` – Pushes item n onto the stack.
  Precond: `not isFull()`; Postcond: `count() = old_count() + 1` and `top() = n`

Figure 2.1: Stack operations

their presence is justified by their use in other methods' contracts.

In general, annotation languages include two features that can be found in postconditions in this example: the possibility of referring to the value of an expression in the pre-state (`old_count()` for `count`), and of referring to the return value (`Result`). The preconditions state that some stack operations cannot be performed on an empty or a full stack, while the postconditions partially specify the functional behaviour of the methods. This is straightforward for `isEmpty` and `isFull`. For `push` the postcondition ensures that the element at the top of the stack on exit is indeed the pushed value, and the stack count is increased with respect to its initial value; for `top` the contract simply states that the count is decreased. It is implicit that the stack remains unmodified, with the exception of its top element when performing a push or pop operation.

Although program verification based on preconditions and postconditions predates design by contract by decades, it has been revitalized by the growing popularity of the latter and the advent of specification languages like JML [LBR99], intended to be used by different tools ranging from dynamic checking to test-case generation, static analysis and verification. In contract-based program verification each procedure C is annotated with a contract (Precond: P; Postcond: Q); checking its correctness amounts to establishing the validity of the Hoare triple $\{P\}\, C\, \{Q\}$ [Hoa69]. A program is correct if all its constituent annotated procedures are correct. The verification process follows the mutually recursive nature of programs: in proving the correctness of procedure f that invokes procedure g, one simply assumes the correctness of g with respect to its contract. In a deductive framework, correctness of a program can be established by the following steps.

1. Annotating the source code with specifications in the form of contracts (for every procedure / function / method) and invariant information for loops;

2. Generating from the code, with the help of a verification conditions generator tool (VCGen for short), a set of first-order proof obligations (verification conditions, VCs), whose validity will imply the correctness of the code; and

3. Discharging the verification conditions using a proof tool. If all VCs are valid then the program is correct.

2.2.3 Bounded Stack: Specification

We use the bounded stack example to illustrate the differences between the verified development of a small software module in SPARK and C/ACSL. We first discuss a few modifications of the typical DbC specification in Figure 2.1. If we think algebraically in terms of the usual stack equations:

$$top(push(n,s)) \quad = \quad n \qquad\qquad pop(push(n,s)) \quad = \quad s$$

Only the first equation is ensured by the contracts of Figure 2.1. Note that the contracts for `push` and `pop` do not state that the methods preserve all the elements in the stack apart from the top element; they simply specify how the number of elements is modified. We will strengthen the specification by introducing a substack predicate, to express the fact that a stack is a substack of another. The notion of substack together with the variation in size allows for a complete specification of the behaviour of these operations. Equally, the contracts for `top`, `isEmpty`, and `isFull` must state that these methods do not modify the stack (i.e. they have no side effects, which is not stated in Figure 2.1).

Additionally, we add to the specification an initialisation function that creates an empty stack. Also, we consider that the operations `count` and `capacity` are not part of the interface of the data type (they are not available to the programmer). In both specification languages `count` and `capacity` will be turned into the logical functions `count_of` and `cap_of` that exist only at the level of annotations, and are not part of the program. These logical functions are sometimes also called hybrid functions because they read program data. In ACSL they are declared inside an `axiomatic` section at the beginning of the file. Note that no definition or axioms can be given at this stage for the logical functions.

In SPARK (as in Ada) the specification and implementation of a package are usually placed in two separate files: the package specification (`.ads`) and the package body (`.adb`) containing the implementation. Packages are separately compiled program units that may contain both data and code and provide encapsulation. Figure 2.2 shows the specification file for the Stack package; `StackType` is an abstract type that is used at the specification level and will later be instantiated in a package body. In the package specification a variable `State` of type `StackType` stands for an abstract stack, i.e. an element of the ADT specified. This will be refined in the body into one or more variables of concrete types.

```
package Stack
--# own State: StackType;
is
   --# type StackType is abstract;
   --# function Count_of(S: StackType) return Natural;
   --# function Cap_of(S: StackType) return Natural;
   --# function Substack(S1: StackType; S2: StackType) return Boolean;

   MaxStackSize: constant := 100;

   procedure Init;
   --# global out State;
   --# derives State from;
   --# post Cap_of(State) = MaxStackSize and Count_of(State) = 0;

   function isEmpty return Boolean;
   --# global State;
   --# return Count_of(State) = 0;

   function isFull return Boolean;
   --# global State;
   --# return Count_of(State) = Cap_of(State);

   function Top return Integer;
   --# global State;
   --# pre Count_of(State) > 0;

   procedure Pop;
   --# global in out State;
   --# derives State from State;
   --# pre 0 < Count_of(State);
   --# post Cap_of(State) = Cap_of(State~) and Count_of(State) = Count_of(State~)-1 and
   --#      Substack(State, State~);

   procedure Push(X: in Integer);
   --# global in out State;
   --# derives State from State, X;
   --# pre Count_of(State) < Cap_of(State);
   --# post Cap_of(State) = Cap_of(State~) and Count_of(State) = Count_of(State~)+1 and
   --#      Top(State) = X and Substack(State~, State);
end Stack;
```

Figure 2.2: Stack SPARK specification

```
typedef ... Stack;
Stack st;
/*@ axiomatic Pilha {
  @ logic integer cap_of{L} (Stack st) = ...
  @ logic integer top_of{L} (Stack st) = ...
  @ logic integer count_of{L} (Stack st) = ...
  @ predicate substack{L1,L2} (Stack st) = ...
  @ } */

/*@ requires cap >= 0;
  @ ensures cap_of{Here}(st) == cap  && count_of{Here}(st) == 0;
  @*/
void init (int cap);

/*@ assigns \nothing;
  @ behavior empty:
  @    assumes count_of{Here}(st) == 0;
  @    ensures \result == 1;
  @ behavior not_empty:
  @    assumes count_of{Here}(st) != 0;
  @    ensures \result == 0;
  @*/
int isEmpty (void);

/*@ assigns \nothing;
  @ behavior full:
  @    assumes count_of{Here}(st) == cap_of{Here}(st);
  @    ensures \result == 1;
  @ behavior not_full:
  @    assumes count_of{Here}(st) != cap_of{Here}(st);
  @    ensures \result == 0;
  @*/
int isFull (void);

/*@ requires 0 < count_of{Here}(st);
  @ ensures \result == top_of{Here}(st);
  @ assigns \nothing;
  @*/
int top (void);

/*@ requires 0 < count_of{Here}(st);
  @ ensures cap_of{Here}(st) == cap_of{Old}(st) &&
  @         count_of{Here}(st) == count_of{Old}(st) - 1 &&
  @         substack{Here,Old}(st);
  @*/
void pop(void);

/*@ requires count_of{Here}(st) < cap_of{Here}(st);
  @ ensures cap_of{Here}(st) == cap_of{Old}(st) &&
  @         count_of{Here}(st) == count_of{Old}(st) + 1 &&
  @         top_of{Here}(st) == x && substack{Old,Here}(st);
  @*/
void push (int x);
```

Figure 2.3: Stack ACSL specification: operation contracts

The specification of a bounded stack in ACSL is given in Figure 2.3. For the sake of simplicity we choose to use a global stack variable, but stacks could equally be passed by reference to the C functions. A crucial difference with respect to the SPARK specification is that ACSL has no support for refinement (and neither has C, of course). Thus in the figure the `typedef` declaration is left unfinished. The reader should bear in mind that it will not be possible to reason about stacks without first providing a concrete implementation. Whereas in SPARK/Ada one can have different implementations in different body files for the same package specification file, in C those implementations would have to be obtained using the file in the figure as a template that would be expanded.

Some language features are directly reflected in the two specifications. The SPARK function `Init` will always produce an empty stack with capacity given by the constant `MaxStackSize`, since dynamic allocation is not possible. In the C version it takes the desired stack capacity as argument. Also, we take advantage of SPARK's type system and set the type returned by functions `Cap_of` and `Count_of` to `Natural` rather than `Integer` (since the number of elements cannot be negative). C's type system is much less precise, thus integers are used instead, but note the use of the `integer` ACSL logical type (for logical functions only).

Concerning the two specification languages, different keywords are used to identify preconditions (`pre`, `requires`) and postconditions (`post`, `ensures`), as well as the return values (`return`, `\result`). Also, ACSL offers the possibility of using optional behaviours in specifications, which permits the association of more than one contract to a function. For instance the behaviour `empty` (resp. `not_empty`) of function `isEmpty` corresponds to the precondition that the current count is zero (resp. not zero), specified with an assumes clause. Behaviours allow for more readable specifications and for more structured sets of VCs.

C functions may in general have side effects, whereas SPARK functions are by definition pure: they are not allowed to modify the global state or to take parameters passed by reference. Thus the SPARK functions `isEmpty`, `isFull`, and `top` are not allowed to modify the state of the stack, which is an improvement (obtained for free) with respect to the contracts in Figure 2.1. In ACSL functions can be annotated with frame conditions that specify the modified parts of the state (variables, structure fields, array elements, etc). The frame conditions of the above three pure functions are written `assigns \nothing`. Appropriate verification conditions are generated to ensure the validity of each frame condition.

A consequence of the previous difference is that SPARK allows for program functions to be used in assertions, whereas in ACSL this is forbidden because of the possibility of side effects. This is reflected in different treatments of the `Top` function in both languages: in the SPARK specification `Top` is a program function and it is used in the postcondition of `Push`, whereas in ACSL a new logical function `top_of` is used; its relation with the `top` program function is established by a postcondition of the latter. In addition to logical / hybrid functions, ACSL offers the possibility of having predicates to be used in annotations; they may be either defined or else declared and their behaviour described by means of axioms. In SPARK a predicate must be declared as a logical function that returns a boolean. This is reflected in the declarations of `substack` in both languages.

In ACSL it is possible to refer to the value of an expression in a given program state, which is extremely useful in any language with some form of indirect memory access. In fact, all hybrid functions and predicates must take as extra arguments a set of state labels in which the value of the parameters are read, even if this set is singular. Thus, for instance, whereas in SPARK the `substack` predicate takes two stacks as arguments, and is invoked (in the postconditions of `Pop` and `Push`) with arguments `State` and `State~`, where the latter refers to the state of the stack in the pre-state, the ACSL version takes as arguments a single stack variable *st* and two state labels L_1, L_2 , with the meaning that the value of *st* in state L_1 is a substack of the value of *st* in state L_2. It is then invoked in annotations making use of predefined program labels Here (the current state) and Old (the pre-state in which the function was invoked).

In SPARK the procedures `Init`, `Pop`, and `Push` have data flow annotations with the meaning that the state of the stack is both read and modified, and the new state depends on the previous state (and for `Push` also on the argument `X`). In functions, the `--# global State;` data flow annotation simply means that these functions read the state of the stack. At this abstract level of development, it is not possible to specify with either SPARK data flow annotations or ACSL frame conditions that the procedures do not modify some part of the state (e.g. `pop` and `push` preserve the capacity). This has then to be done using postconditions.

Reasoning about Specifications in SPARK. A major difference between both languages is that in SPARK it is possible to reason in the absence of concrete implementations. To illustrate this, we will define a procedure that swaps the values of two variables using a stack. The relevant package and body are shown in Figure 2.4. Running the SPARK Examiner on this file produces 9 verification conditions, of which, after running the SPARK Simplifier, only one is left unproved. This VC is generated from the postcondition of `Swap`, which is only natural since we haven't given a definition of `substack`.

The SPARK Simplifier allows users to supply additional rules and axioms, in the FDL logical language, in a separate file. The following SPARK rule states that two equally sized substacks of the same stack have the same top elements.

```
ss_rule(1) : stack__top(S1) = stack__top(S2) may_be_deduced_from
  [stack__count_of(S1) = stack__count_of(S2), stack__substack(S1,S3), stack__substack(S2,S3)].
```

Unfortunately, even with this rule, the Simplifier fails to automatically discharge the VC, so the user would be forced to go into interactive proof mode (using the SPARK Proof Checker) to finish verifying the program. Alternatively, the following rule allows the Simplifier to finish the proof automatically:

```
with Stack;
--# inherit Stack;
package SSwap is
  procedure Swap(X, Y: in out Integer);
   --# global in out Stack.State;
   --# derives Stack.State, X, Y from Stack.State, X, Y;
   --# pre Stack.Count_of(Stack.State) <= Stack.Cap_of(Stack.State)-2;
   --# post X = Y~ and Y = X~;
end SSwap;

package body SSwap is
  procedure Swap(X, Y: in out Integer)
   is
   begin
      Stack.Push(X); Stack.Push(Y);
      X := Stack.Top; Stack.Pop;
      Y := Stack.Top; Stack.Pop;
   end Swap;
end SSwap;
```

Figure 2.4: Swap using a stack

```
ss_rule(3) : stack__top(S1) = stack__top(S2) may_be_deduced_from
  [stack__count_of(S3) = stack__count_of(S2)+1, stack__count_of(S1) = stack__count_of(S3)-1,
   stack__substack(S1,S3), stack__substack(S2,S3)].
```

This also illustrates a technique that we find very useful with the Simplifier: writing special purpose rules that follow the structure of the computation. In this example we have simply mentioned explicitly the intermediate stack S_3 that the state goes through betwen S_2 and S_1. This is often sufficient to allow the Simplifier to discharge all VCs without the need for interactive proof.

2.2.4 Bounded Stack: Implementation / Refinement

Figure 2.5 shows a fragment of the stack package implementation, including the definition of the state and the definition of the `Push` procedure. The corresponding fragment in C is given in Figure 2.6. The state is in both cases defined as a set of two integer variables (for the size and capacity) together with an array variable. In SPARK a range type `Ptrs` is used, which is not possible in C.

In C we simply fill in the template of Figure 2.3 without touching the annotations. We consider a straightforward implementation of bounded stacks as structures containing fields for the capacity and size, as well as a dynamically allocated array. This requires providing, in addition to the C function definitions, appropriate definitions of the logical functions `cap_of`, `top_of`, and `count_of`, as well as of the predicate `substack`. `count_of` and `cap_of` simply return the values of structure fields. The most sophisticated aspect is the use of a universal

```
package body Stack
--# own State is Capacity, Ptr, Vector;
is
   type Ptrs is range 0..MaxStackSize;
   subtype Indexes is Ptrs range 1..Ptrs'Last;
   type Vectors is array (Indexes) of Integer;

   Capacity: Ptrs := 0;
   Ptr: Ptrs := 0;
   Vector: Vectors := Vectors'(Indexes => 0);

   procedure Push(X: in Integer)
   --# global in out Vector, Ptr;
   --#        in Capacity;
   --# derives Ptr from Ptr & Vector from Vector, Ptr, X & null from Capacity;
   --# pre Ptr < Capacity;
   --# post Ptr = Ptr~ + 1 and Vector = Vector~[Ptr => X];
   is
   begin
      Ptr := Ptr + 1;
      Vector(Ptr) := X;
      --# accept F, 30, Capacity, "Only used in contract";
   end Push;
```

```
stack_rule(1) : cap_of(S) may_be_replaced_by fld_capacity(S) .
stack_rule(2) : count_of(S) may_be_replaced_by fld_ptr(S) .
stack_rule(3) : count_of(X) = count_of(Y) - Z may_be_replaced_by fld_ptr(Y) = fld_ptr(X) + Z.
stack_rule(4) : count_of(X) = count_of(Y) + Z may_be_replaced_by fld_ptr(X) = fld_ptr(Y) + Z.
stack_rule(5) : count_of(S) = cap_of(S) may_be_replaced_by fld_ptr(S) = fld_capacity(S).
stack_rule(6) : substack(X, Y) may_be_deduced_from
  [V=fld_vector(X), Z=fld_ptr(X)+1, Z=fld_ptr(Y), fld_vector(Y)=update(V, [Z], N)].
stack_rule(7) : substack(X, Y) may_be_deduced_from
  [fld_vector(X)=fld_vector(Y), fld_ptr(X)<fld_ptr(Y)].
stack_rule(8) : stack__top(X) = Y may_be_deduced_from
  [fld_vector(X) = update(Z, [fld_ptr(X)], Y)] .
```

Figure 2.5: Stack SPARK implementation (fragment) and user-provided rules

```
typedef struct stack {
  int capacity;
  int size;
  int *elems;
} Stack;

int x, y;
Stack st;

/*@ axiomatic Pilha {
  @ logic integer cap_of{L} (Stack st) = st.capacity;
  @ logic integer top_of{L} (Stack st) = st.elems[st.size-1];
  @ logic integer count_of{L} (Stack st) = st.size;
  @ predicate substack{L1,L2} (Stack st) = \at(st.size,L1) <= \at(st.size,L2) &&
  @  \forall integer i; 0<=i<\at(st.size,L1) ==> \at(st.elems[i],L1) == \at(st.elems[i],L2);
  @ predicate stinv{L}(Stack st) =
  @  \valid_range(st.elems,0,st.capacity-1) && 0 <= count_of{L}(st) <= cap_of{L}(st);
  @ } */

/*@ requires count_of{Here}(st) < cap_of{Here}(st) && stinv{Here}(st);
  @ ensures cap_of{Here}(st) == cap_of{Old}(st) && count_of{Here}(st) == count_of{Old}(st)+1
  @  && top_of{Here}(st) == x && substack{Old,Here}(st) && stinv{Here}(st);
  @*/
void push (int x) {
  st.elems[st.size] = x;
  st.size++;
}

/*@ ensures x == \old(y) && y == \old(x);
  @*/
swap() {
  init(3); push(x);  push(y); x = top(); pop(); y = top(); pop();
}
```

Figure 2.6: Stack C implementation (extract) and test function (swap)

quantifier in the definition of `substack`. Note also the use of the operator `\at` to refer to the value of a field of a structure variable in a given program state (not required when a single state label is in scope – it is implicit).

SPARK on the other hand has explicit support for refinement. Thus contracts can be written at a lower level using the state variables, as exemplified by the `Push` procedure. Since there are no logical definitions as such in SPARK, the functions `cap_of` and `count_of` will be handled by the user rules `stack_rule(1)` and `stack_rule(2)` that can be applied as rewrite rules in both hypotheses and conclusions. The user rules 3 to 5 are auxiliary rules; their presence illustrates the limitations of the Simplifier in applying the previous 2 rewrite rules.

Refinement Verification in SPARK. Invoking the SPARK examiner with both package and body files will produce a set of verification conditions, establishing a correspondence between specification and implementation contracts in the classic sense of refinement: given a procedure with specification precondition P_s (resp. postcondition Q_s) and body precondition P_b (resp. postcondition Q_b), the VCs $P_s \implies P_b$ and $Q_b \implies Q_s$ will be generated, together with conditions for correctness of the procedure's body with respect to the specification (P_b, Q_b).

A crucial refinement aspect of our example has to do with the `substack` predicate. Note that there is no mention of the predicate at the implementation level, so we must now provide rules for inferring when a stack is a substack of another. Writing a rule based on the use of a quantifier (as we did in ACSL) would not help the Simplifier (although it could be used for interactive proof), thus we provide instead rule (6) for the specific case when X is a substack of Y that contains only one more element (`fld_vector` and `fld_ptr` correspond to the fields `Vector` and `Ptr` respectively in the stack body), and rule (7) regarding the case of two stacks represented by the same vector with different counters. These basically describe what happens in the `push` and `pop` operations.

In these rules we make use of the fact that SPARK arrays are logically modeled using the standard theory of arrays [Rey79], accessed through the `element` and `update` operations. In particular the expression `update(V, [Z], N)` denotes the array that results from array `V` by setting the contents of the position with index `Z` to be `N`. Rule (8) concerns the top of a stack after an update operation at the `ptr` position. With these rules the Simplifier is able to discharge all VCs.

Verification of C code. Our C/ACSL file now contains a full implementation of the stack operations, based on the previously given contracts. Let us add to this a `swap` function (also shown in Figure 2.6). Running Frama-C on this file will generate verification conditions that together assert that the code of the stack operations and of the `swap` function conforms to their respective contracts. 38 VCs are generated, only 4 of which, labelled "pointer dereferencing", are not discharged automatically. These are safety conditions, discussed below.

Safety Checking. Being able to write exception-free code is a very desirable feature in embedded and critical systems. In the stack example this is relevant for array out-of-bounds access, and again the two languages offer different approaches. An important feature of SPARK is that, using proof annotations and automatically generated safety conditions, programs can be shown statically not to cause runtime exceptions. The expression <u>runtime checks</u> (or <u>safety conditions</u>) designates VCs whose validity ensures the absence of runtime errors.

In the SPARK implementation the domain type of the array is a range type (as are the other state variables), which in itself precludes out-of-bounds access. The runtime errors that may occur concern precisely the range types: every use of an integer expression (in particular in assignments and array accesses) will generate conditions regarding the lower and upper bounds of the expression. For instance the instruction `Ptr := Ptr + 1` in the `Push` procedure generates a VC to check that `ptr + 1` lies within the range of type `Indexes`. Such conditions are generated and automatically discharged in both the `swap` and the refinement verification in a completely transparent way.

ACSL on the other hand treats array accesses (and pointer dereferencing in general) through special-purpose annotations. This is motivated by the very different nature of arrays in C – in particular they can be dynamically allocated and no range information is contained in their types. A `valid_range` annotation in a function precondition expresses that it is safe for the function to access an array in a given range of indexes. It should also be mentioned that a memory region separation assumption is used by default when reasoning about arrays.

Frama-C automatically introduces verification conditions for checking against out-of-bound accesses, thus the 4 VCs left unproved in our example. In order to address this issue we create a new predicate `stinv` that expresses a safety invariant on stacks (the count must not surpass the capacity, and array accesses should be valid within the range corresponding to the capacity). It suffices to include this predicate as precondition and postcondition in all operation contracts (with the exception of the precondition of `init`) for the safety conditions to be automatically discharged. The modifications are already reflected in Figure 2.6.

2.2.5 Conclusion

We are of course comparing two very different toolsets, one for a language with dynamic memory and ‘loose’ compilation, and another for a memory-bounded language with very strict compilation rules and side-effects explicitly identified in annotations (not to mention the refinement aspect). From our experience with SPARK and the study of published case studies the Simplifier does a very good job of automatically discharging safety conditions. The Simplifier has been compared with SMT solvers, and the relative advantages of each discussed [JES07].

While it would be unfair to compare SPARK with Frama-C in terms of the performance of safety checking (in particular because SPARK benefits from the strict rules provided by

Ada regarding runtime exceptions), we simply state that safety-checking ACSL specifications requires an additional effort to provide specific safety annotations, whereas in SPARK runtime checks are transparently performed. On the other hand a general advantage of Frama-C is the multi-prover aspect of the VCGen: one can effortlessly export VCs to different provers, including tools as diverse as SMT solvers and the Coq [dt04] proof assistant. Finally, it is important to remark that unlike SPARK, to this date Frama-C has not, to the best of our knowledge, been used in large-scale industrial projects.

The situation changes significantly when other functional aspects are considered. Take this example from the Tokeneer project, a biometric secure system implemented in SPARK and certified according to the Common Criteria higher levels of assurance (`http://www.adacore.com/tokeneer`). We were quite surprised to find that the Simplifier is unable to prove `C1` from `H20`:

```
H20: element(logfileentries__1, [currentlogfile]) =
     element(logfileentries, [currentlogfile]) + 1 .
-> C1: element(logfileentries, [currentlogfile]) -
       element(logfileentries__1, [currentlogfile]) = - 1 .
```

Simple as it is, our case study has shown that the Simplifier's ability for reasoning with logical functions and user-provided rules is quite limited. Also, our experiences with more 'algorithmic' examples involving loop invariants show that Frama-C is quite impressive in this aspect. For instance fairly complex sorting algorithms, involving nested loops and assertions with quantification, can be checked in Frama-C in a completely automatic manner, with no additional user-provided axioms or rules. In this respect it is our feeling that the SPARK technology needs to be updated or complemented with additional tools.

To sum up our findings, the effort that goes into verifying safe runtime execution is smaller in SPARK, whereas the situation seems to be reversed when the specification and automatic verification of other functional aspects is considered.

One aspect that our running example has not illustrated is related to aliasing. Reasoning about procedures with parameters passed by reference is typically difficult because such a procedure may access the same variable through different lvalues, for instance a procedure may access a global variable both directly and through a parameter. In SPARK such situations are rejected by the examiner after data-flow analysis, so verification conditions are not even generated.

In C such programs are of course considered valid, but note that these situations can only be created by using pointer parameters, and it is possible to reason about such functions with pointer-level assertions. For instance, a function that takes two pointer variables may have to be annotated with an additional precondition stating that the values of the pointer parameters (not the dereferenced values) are different. We have stressed the importance of the use of state labels in ACSL; for reasoning about dynamic structures, serious users of Frama-C will also want to understand in detail the memory model underlying ACSL and the associated separation assumptions, which is out of our scope here.

Finally, we should mention that other tools are available for checking C code, such as VCC [CDH$^+$]. Many other verification tools exist for object-oriented languages; Spec# [BRS05] is a good example.

3 mSPARK: A Subset of SPARK

In this chapter we present the subset of SPARK which we have formalized and we present the methodology that we used to develop this subset of the language. We also propose a toolchain and describe in detail how the development of the interpreter in Haskell has helped us, thanks to its strongly typed type system.

Even if SPARK is a relatively small programming language, it is still a challenge to fully formalize it. In the beginning we tried to work with a larger subset of the language but unfortunately we had to leave some features of the language out of our formalization because of time constraints.

3.1 Developing mSPARK

In this section we present the methodology being used to develop mSPARK (which we intend to extend to future versions of the language, not just Hibana), its tools and some of the work we aim to achieve in the near future.

3.1.1 The Development Methodology of the mSPARK Language & Semantics

Hibana, written as seen in Figure 3.1, is the initial version of mSPARK, a modest subset of the SPARK language. Although modest, we believe that this subset is a step forward in the formalization of a "real" programming language by providing features that are not usually considered in While-like languages. One such thing is the type system that is available in Hibana. SPARK is a highly structured and strongly typed language with many restrictions and specifics in its type system. The existence of ranges in the types provides a challenge for its formalization. Furthermore, we aim at providing an axiomatic semantics for the language and its respective VCGen.

While at this stage in the project it is not feasible to develop a mechanically verified VCGen, as has already been done for a While-like language (for instance, by Homeier [HM94]), we will initially provide pen and paper proofs of the properties of our axiomatic semantics. Also, we have checked that our operational semantics (which is writen in a big-step/natural style) is, at least, properly typed by writing the rules in Haskell [Jon03]. Haskell provides a straightforward way to represent the rules as an embedded programming language (EPL). Besides ensuring that we are not making fundamental mistakes regarding the typing of rules,

火花

Figure 3.1: Hibana written in Japanese.

the EPL also allows us to implement an evaluator/interpreter for our programming language.

In Figure 3.2 we represent the approach that we have taken in the development of Hibana. As expected, we have started with the SPARK language, chosen a subset of the language, which we named mSPARK, and then we proceeded to specify the operational semantics of this subset using a standard mathematical notation (Natural Semantics). This specification was then coded in Haskell as an EPL and provided feedback on the typing of our rules.

Having a stable version of the natural semantics, we then proceed to specify the axiomatic semantics (using a Hoare-like Logic [Hoa69]). When specifying the axiomatic semantics we may find it necessary to add extra information to the Natural Semantics; also, adding things to the Natural Semantics may have impact on the axiomatic semantics.

While it may not be obvious from the figure, the development of the axiomatic semantics and of the VCGen can be done in parallel. For each rule we write in the axiomatic semantics we are able to further the development of the VCGen specification. As before, while writing the VCGen we may notice that we need more information from the axiomatic semantics and this may lead to changes on everything that has been done before. The implementation of the VCGen is also a parallel task that can be written at the same time as the VCGen is specified.

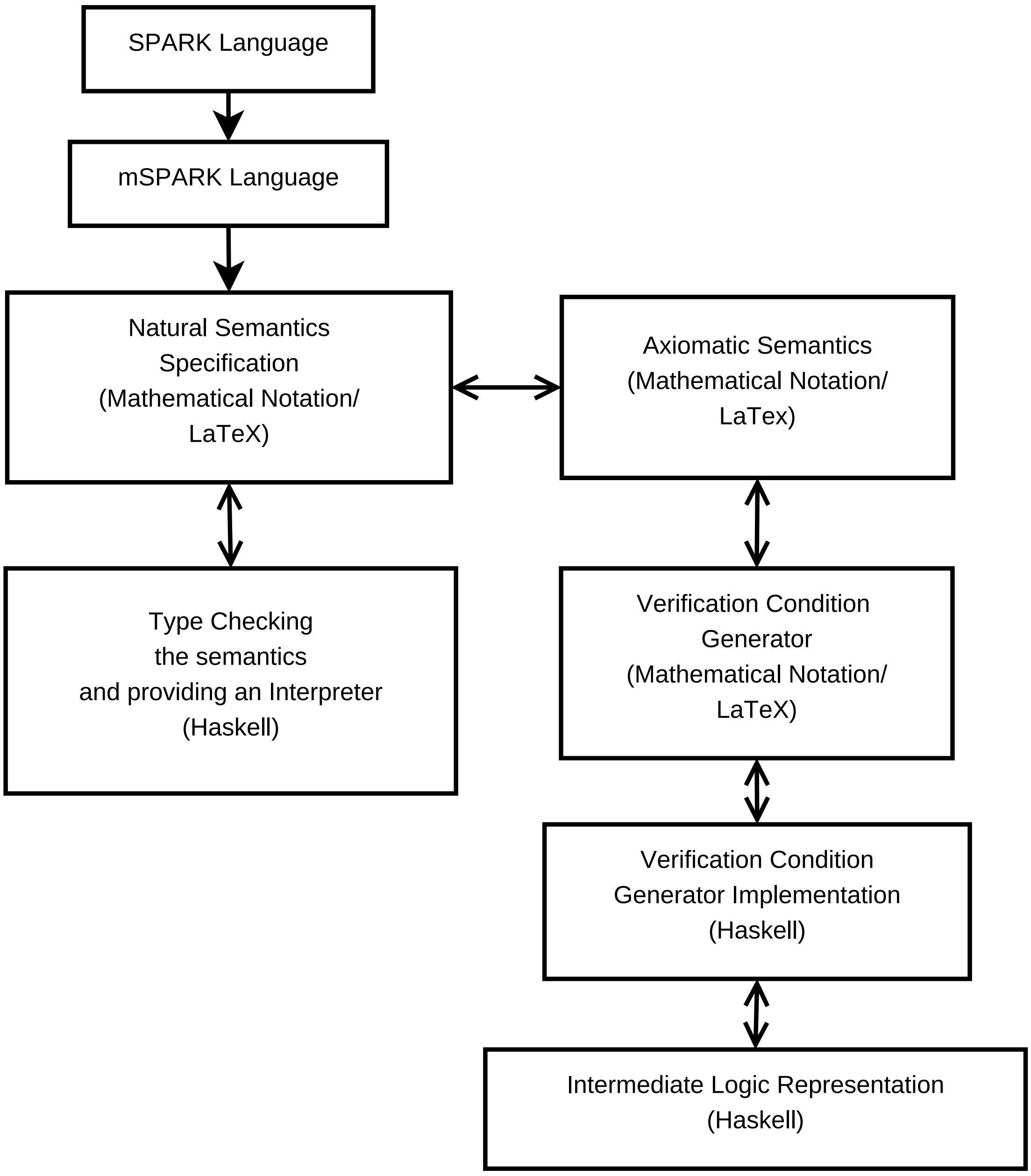
A problem of this development methodology is that it did not make much sense to reason about a language, mSPARK, until a minimum set of constructs were already established. So, in the beginning, the development of the natural semantics was in fact a bottleneck in the development cycle.

3.1.2 The mSPARK Programming Language

As previously stated, mSPARK is a subset of SPARK. Given this, we present in Table 3.1 the list of features that can be found in the SPARK programming language and their status in our subset. Items marked with ✓are fully implemented, items marked with X are not implemented, and items with ± are partially implemented. We also added a column to the table, justifying some of our options.

The list is only related to the features of the programming language itself. In Section 3.1.3 we will present the tools that support our methodology.

Most features that are not implemented are in our view not as important as the features we



SPARK Language
mSPARK Language
Natural Semantics Specification (Mathematical Notation/ LaTeX)
Axiomatic Semantics (Mathematical Notation/ LaTex)
Type Checking the semantics and providing an Interpreter (Haskell)
Verification Condition Generator (Mathematical Notation/ LaTeX)
Verification Condition Generator Implementation (Haskell)
Intermediate Logic Representation (Haskell)

Figure 3.2: The mSPARK methodology.

$$\text{VCGen} \xrightarrow{\text{Sound w.r.t}} \text{Program Logic} \xrightarrow{\text{Sound w.r.t}} \text{Operational Semantics}$$

$$\text{Operational Semantics} \xrightarrow{\text{Type checks in}} \text{Haskell}$$

Figure 3.3: mSPARK validation.

chose. The features that were chosen were based on our experience in developing programs in SPARK and other programming languages and also based on what we deemed to be more important and interesting to formalize.

Feature	mSPARK (Hibana)	Comments
Basic commands	✓	
Basic control structures	✓	
Extended control structures	✓	
Basic data types	±	Some types, such as fixed and floating-point and modular arithmetic are not included, for now, in the language.
Enumeration types	✓	
Range types/subtypes	✓	
Type casting	X	
Record types	X	
Subprograms	±	We allow parameterless procedures with frame conditions and pure functions (with and without parameters).
Named Parameters	X	
Assertion annotations	✓	
Data-flow annotations	±	We only use data-flow annotations for frame conditions.
Packages	±	We use a different syntax from SPARK. Also, we do not support some of SPARK's features.
Object Orientation	X	An important addition to future iterations of the language. SPARK has only partial support for it.
Refinement	X	One of the most important features we have on our "wishlist". The most important part will be to have the axiomatic semantics and VCGen for this feature.
RavenSPARK	X	Important feature for the future. We intend to have the operational and axiomatic semantics for this subset of concurrency.
Generics	X	SPARK's team has been struggling with this feature for some time. We would like to formally specify this in our language.

Table 3.1: Features of the mSPARK programming language.

3.1.3 The mSPARK Tools

mSPARK Hibana is a subset of SPARK. This means that all mSPARK Hibana programs have a corresponding SPARK program but not all SPARK programs have a corresponding mSPARK program. We propose the toolchain in Fig. 3.4 as an adapter layer from SPARK to mSPARK, for a subset of SPARK that corresponds to mSPARK Hibana[1].

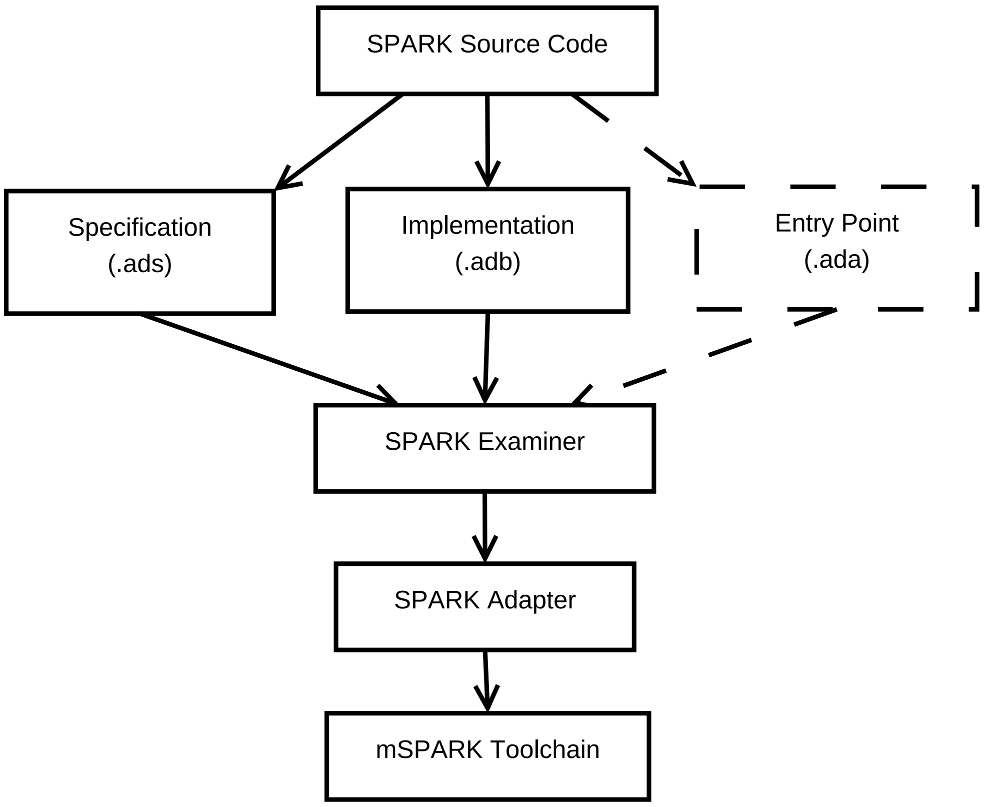


Figure 3.4: Adaptation layer from SPARK to mSPARK.

It should be noted that, for now, we depend on the SPARK Examiner to check if we have valid source code. As our language is further developed, the use of SPARK's original Examiner will become deprecated and there will be a need to develop our own Examiner to check our subset, with the enhancements we plan to introduce in the language.

As we show in Fig. 3.5, we may not need/want to use SPARK and the adapter but instead we may want to use the mSPARK language directly. Either way, what is important is to have the intermediate code representation for our source code. With that intermediate representation we can execute/animate it with the EPL Interpreter (if the source code contains an executable component) and, using the same intermediate representation, we can supply it to the VCGen.

[1]It is important to distinguish that this is true for this version of the language, Hibana, but may not be applicable to future versions of the language.

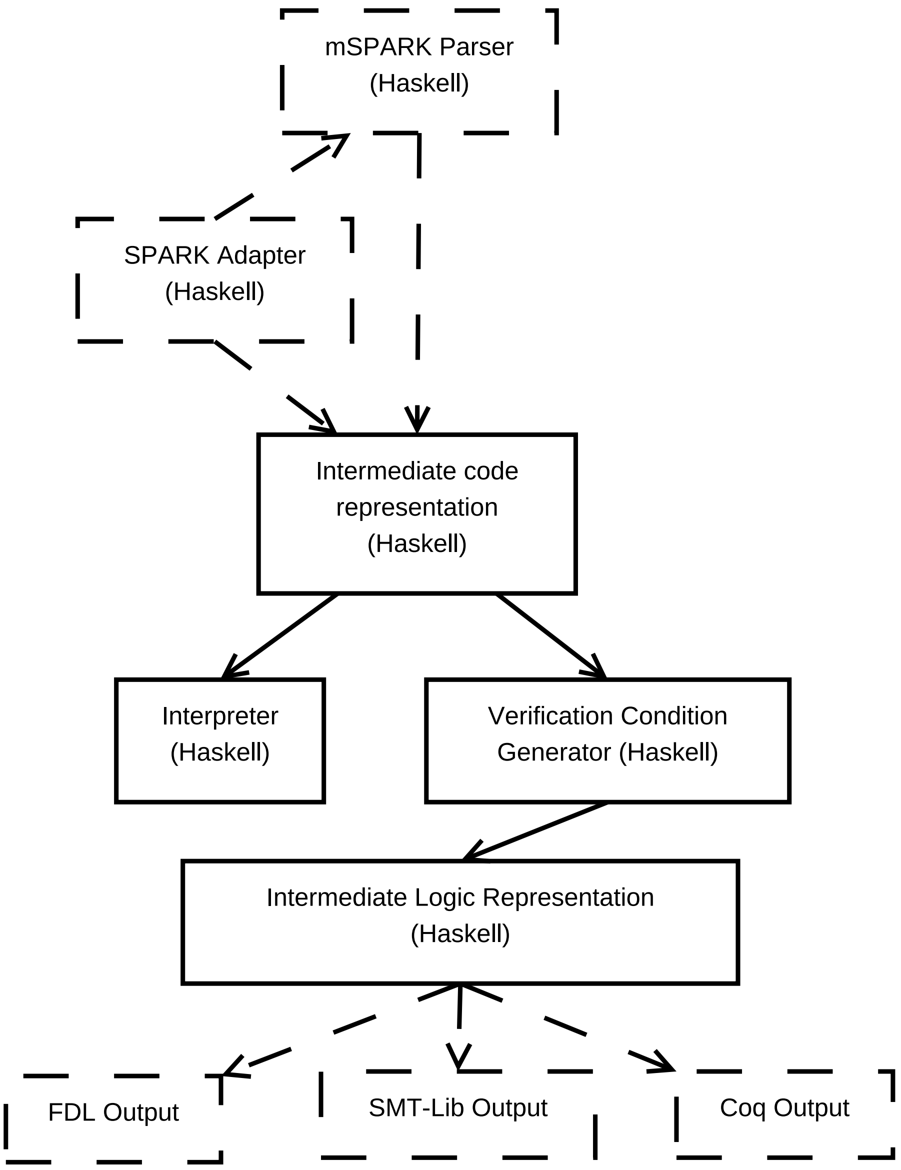


Figure 3.5: The mSPARK toolchain.

3.2 Syntax, Static Semantics and Natural Semantics of mSPARK

In this section we present the syntax and operational semantics of the mSPARK programming language.

The syntax will be presented in abstract and concrete forms. We relate productions in both syntactic forms by their names. These names will also be used in parts of the operational semantics, particularly when defining the meta-variables in Section 3.2.3.

We also present the semantics of mSPARK. The static semantics of the language is given an informal treatment for the most part. We chose to do this because, while it is necessary to have some description of mSPARK's static semantics, given the limited time, we chose to focus on the natural semantics since it is essential to prove the soundness of the program logic (although we still need to assume certain things from the static semantics, as we shall see in Chapter 4).

3.2.1 Syntax of mSPARK Hibana

In the abstract syntax (Figure 3.6) the typing that our constructs have is clearer than what we have in the concrete syntax. One of the aims of providing the abstract syntax is exactly that; to determine the classes of our constructs and how they are typed.

In the concrete syntax we say what syntactic constructions are valid for our language. Note that this addresses only the syntactic aspects. While in the abstract syntax we identified types and classes of constructs, in the concrete syntax we are only considering parsing issues. One case where this is especially relevant is in the expressions (denoted by $< e >$).

In the abstract syntax we describe the types of expressions that are possible to be constructed: boolean expressions $< be >$, arithmetic expressions $< ae >$ and constant arithmetic expressions $< cae >$. In this syntax, these expressions are typed only for those particular types.

In the concrete syntax we can construct any kind of expression, even if they are not properly typed, as long as it obeys the syntactic laws. This issue, among others, makes it necessary for a static semantics that enforces that the parsing result is a correct expression of the expected type.

It is also presented an example of source code written in mSPARK in Code 1. We shall revisit this example in Section 3.3.

3.2.2 Informal Static Semantics of mSPARK

In this section we present an informal static semantics for mSPARK. The static semantics shall enforce restrictions upon the syntactic constructs so that we guarantee that our environment is well formed. This means enforcing aspects of the abstract syntax in the concrete syntax as well as defining properties of the programming language.

Scope The semantics for our scoping rules is known as "static scope". The scoping rules are applied to variables and subprograms.

- Variables declared at **Variables** are considered global variables and are at the outermost scope; they are visible at all levels, unless overridden;
- Subprograms may have local variables; these local variables are at an inner scope and shall override variables in outer scopes, such as global variables, with the same identifier;

$\langle \mathit{program} \rangle$ ::= TYPES $\langle \mathit{dt} \rangle$ VARIABLES $\langle \mathit{dv} \rangle$ SUBPROGRAMS $\langle \mathit{dsp} \rangle$ EXECUTE $\langle \mathit{s} \rangle$ END

$\langle \mathit{s} \rangle$ ::= NULL
  | var ‘`:=`’ $\langle \mathit{e} \rangle$
  | IF $\langle \mathit{be} \rangle$ THEN $\langle \mathit{s} \rangle$ (ELSIF $\langle \mathit{be} \rangle$ THEN $\langle \mathit{s} \rangle$)* [ELSE $\langle \mathit{s} \rangle$]END IF
  | WHILE $\langle \mathit{assert} \rangle$ $\langle \mathit{be} \rangle$ LOOP $\langle \mathit{s} \rangle$ END LOOP
  | LOOP $\langle \mathit{assert} \rangle$ EXIT WHEN $\langle \mathit{be} \rangle$ $\langle \mathit{s} \rangle$ END LOOP
  | LOOP $\langle \mathit{assert} \rangle$ $\langle \mathit{s} \rangle$ EXIT WHEN $\langle \mathit{be} \rangle$ END LOOP
  | FOR IDv IN IDt RANGE $\langle \mathit{dr} \rangle$ LOOP $\langle \mathit{assert} \rangle$ $\langle \mathit{s} \rangle$ END LOOP

$\langle \mathit{var} \rangle$ ::= IDv List$\langle \mathit{e} \rangle$

$\langle \mathit{dt} \rangle$ ::= TYPE IDt IS $\langle \mathit{dr} \rangle$ [$\langle \mathit{dt} \rangle$]
  | SUBTYPE IDt IS IDt $\langle \mathit{dr} \rangle$ [$\langle \mathit{dt} \rangle$]

$\langle \mathit{dv} \rangle$ ::= List$\langle \mathit{IDv} \rangle$ ‘`:`’ IDt [‘`:=`’ $\langle \mathit{e} \rangle$] [$\langle \mathit{dv} \rangle$]

$\langle \mathit{dsp} \rangle$ ::= FUNCTION IDf List$\langle \mathit{param} \rangle$ RETURN ID IS $\langle \mathit{assert} \rangle$ $\langle \mathit{assert} \rangle$ [$\langle \mathit{dv} \rangle$] [$\langle \mathit{dsp} \rangle$] BEGIN $\langle \mathit{s} \rangle$
    RETURN $\langle \mathit{e} \rangle$ END IDf [$\langle \mathit{dsp} \rangle$]
  | PROCEDURE IDp IS $\langle \mathit{assert} \rangle$ $\langle \mathit{assert} \rangle$ $\langle \mathit{global} \rangle$ [$\langle \mathit{dv} \rangle$] [$\langle \mathit{dsp} \rangle$] BEGIN $\langle \mathit{s} \rangle$ END IDp [$\langle \mathit{dsp} \rangle$]

$\langle \mathit{global} \rangle$ ::= List$\langle \mathit{IDv} \rangle$

$\langle \mathit{op} \rangle$ ::= + | - | * | /

$\langle \mathit{rop} \rangle$ ::= ‘`<`’|‘`<=`’|‘`>`’|‘`>=`’|‘`=`’|‘`/=`’

$\langle \mathit{bop} \rangle$ ::= AND | OR

$\langle \mathit{e} \rangle$ ::= $\langle \mathit{be} \rangle$
  | $\langle \mathit{ae} \rangle$
  | $\langle \mathit{cae} \rangle$

$\langle \mathit{be} \rangle$ ::= $\langle \mathit{bool} \rangle$ | $\langle \mathit{fcall} \rangle$
  | $\langle \mathit{ae} \rangle$ $\langle \mathit{rop} \rangle$ $\langle \mathit{ae} \rangle$
  | NOT $\langle \mathit{be} \rangle$
  | $\langle \mathit{be} \rangle$ $\langle \mathit{bop} \rangle$ $\langle \mathit{be} \rangle$

$\langle \mathit{ae} \rangle$ ::= $\langle \mathit{number} \rangle$ | $\langle \mathit{var} \rangle$ | $\langle \mathit{fcall} \rangle$
  | $\langle \mathit{ae} \rangle$ $\langle \mathit{op} \rangle$ $\langle \mathit{ae} \rangle$

$\langle \mathit{cae} \rangle$ ::= $\langle \mathit{number} \rangle$
  | $\langle \mathit{cae} \rangle$ $\langle \mathit{op} \rangle$ $\langle \mathit{cae} \rangle$

$\langle \mathit{fcall} \rangle$ ::= IDf List$\langle \mathit{e} \rangle$

$\langle \mathit{assert} \rangle$ ::= $\langle \mathit{be} \rangle$
  | NOT $\langle \mathit{assert} \rangle$
  | $\langle \mathit{assert} \rangle$ ($\Longrightarrow$ | $\Longleftrightarrow$) $\langle \mathit{assert} \rangle$
  | (FORALL | EXISTS) IDv $\langle \mathit{assert} \rangle$

Figure 3.6: mSPARK Abstract Syntax

```
⟨program⟩ ::= TYPES ⟨dt⟩ VARIABLES ⟨dv⟩ SUBPROGRAMS ⟨dsp⟩ EXECUTE ⟨s⟩ END

⟨dt⟩ ::= ϵ
   |  TYPE ID IS ⟨dr⟩ EOS [⟨dt⟩]
   |  SUBTYPE ID IS ID ⟨dr⟩ EOS [⟨dt⟩]

⟨dv⟩ ::= ϵ
   |  ID (‘,’ ID)* ‘:’ ID [‘:=’ ⟨e⟩] EOS [⟨dv⟩]

⟨dsp⟩ ::= ϵ
   |  FUNCTION ID [⟨par_list⟩] RETURN ID IS ⟨fun_contract⟩ [⟨dv⟩] [⟨dsp⟩] BEGIN ⟨s⟩ RETURN
      ⟨e⟩ END ID EOS [⟨dsp⟩]
   |  PROCEDURE ID IS ⟨proc_contract⟩ [⟨dv⟩] [⟨dsp⟩] BEGIN ⟨s⟩ END ID EOS [⟨dsp⟩]

⟨par_list⟩ ::= ID (‘,’ ID)* ‘:’ ID EOS [⟨par_list⟩]

⟨fun_contract⟩ ::= PRE ⟨assert⟩ POST ⟨assert⟩

⟨proc_contract⟩ ::= PRE ⟨assert⟩ POST ⟨assert⟩ GLOBAL ID (‘,’ ID)*

⟨assert⟩ ::= [NOT] [(FORALL | EXISTS) ID (‘,’ ID)* ‘.’] ⟨e⟩ [(‘=>’ | ‘<=>’ | AND | OR) ⟨assert⟩]

⟨s⟩ ::= (⟨s_1⟩ EOS)*

⟨s_1⟩ ::= NULL
   |  ⟨var⟩ ‘:=’ ⟨e⟩
   |  IF ⟨e⟩ THEN ⟨s⟩ (ELSIF ⟨e⟩ THEN ⟨s⟩)* [ELSE ⟨s⟩] END IF
   |  WHILE ⟨assert⟩ ⟨e⟩ LOOP ⟨s⟩ END LOOP
   |  LOOP ((⟨assert⟩ EXIT WHEN ⟨e⟩ EOS ⟨s⟩) | (⟨s⟩ EXIT WHEN ⟨e⟩)) END LOOP
   |  FOR ID IN ID RANGE ⟨range_decl⟩ LOOP ⟨assert⟩ ⟨s⟩ END LOOP

⟨var⟩ ::= ID [ ‘(’ ⟨e⟩ (‘,’ ⟨e⟩)* ‘)’]

⟨fcall⟩ ::= ID [ ‘(’ ⟨e⟩ (‘,’ ⟨e⟩)* ‘)’]

⟨e⟩ ::= ⟨be⟩ ((‘<’|‘<=’|‘>’|‘>=’|‘=’|‘/=’) ⟨be⟩)*

⟨be⟩ ::= ⟨ae⟩ ((AND|OR) ⟨ae⟩)*

⟨ae⟩ ::= ⟨sube⟩ ((MUL|DIV) ⟨sube⟩)*

⟨sube⟩ ::= ⟨t⟩ ((ADD|SUB) ⟨t⟩)*

⟨t⟩ ::= ⟨var⟩
   |  ⟨number⟩
   |  ⟨bool⟩
   |  ⟨fcall⟩
   |  NOT ⟨e⟩
   |  ‘(’ ⟨e⟩ ‘)’
```

Figure 3.7: mSPARK concrete syntax.

Code 1 mSPARK Code for Calculating the Maximum of Two Numbers.

```
types

    type Short is range -65536 .. 65536;

variables

    x, y, res : Short;

subprograms

    function Max( a : Short; b : Short ) return Short is
        aux : Short;
    begin

        if a >= b then
            aux := a;
        else
            aux := b;
        end if;
        return aux;

    end Max;

execute

    x := 9001;
    y := 0;
    res := Max( x, y );

end;
```

- Subprograms may also have nested subprograms. Nested subprograms create a new inner scope in which variables with the same identifier may override variables in the outer scopes (parent subprogram and global variables);
- Nested subprograms are only visible to the parent subprograms.

Identifiers Static rules for the naming of identifiers.

- Case insensitive;
- No repeated identifiers at the same scope level. Identifiers for variables, subprograms and types must always be disjoint, at the same scope level;
- Variables at different scopes may have overlapping identifiers;
- Types and subprograms must have unique identifiers at all scope levels, including identifiers for enumeration types.

Expressions Static rules for expressions.

- Expressions at type declarations must be Constant Arithmetic Expressions; this includes values from enumeration types;
- Constant Arithmetic Expressions must be statically checked for range constraints;
- Expressions of type $< e >$ may be of any subtype;
- Expressions must always type to the desired expression type. If what is expected is a boolean expression ($< be >$), than the type of the expression must be boolean; the same applies to the other types of expressions;
- Expressions can not yield side effects; this shall be a consequence of how expressions are written and of the static restrictions applied to function declarations;
- Expressions in assignments must be checked if the types used in the identifiers of the expression match the type of the variable being assigned, e.g we can not assign Weight to Height or Boolean to Int;
- Type casting is not supported, for now. This refers to explicit type casting but also implicit type casting, e.g. subtypes can not be promoted automatically to base types.

Functions Static rules for functions.

- Functions can only read variables from the outer scopes; writing to variables of outer scopes if prohibited;
- Function parameters can not be used as regular local variables. They can only be read;

This static semantics, although not rigorous nor formal, is enough to guarantee some properties that we want to ensure in our language, such as initial environments being well formed. Preservation of well formedeness will be guaranteed by the Natural Semantics.

3.2.3 Types, Environment, Functions and Meta-variables

In this section we introduce some definitions that are required for the specification of the NS of mSPARK in the next section. In particular, we define the structure of environments to be used, the data types used in our semantics, the auxiliary functions for the semantics, the evaluation functions, the transition relations and the meta-variables that range over the data types of the semantics.

A note on notation The types that we will present here are represented in a Haskell-like notation, for the most part. For instance, this can be seen in the "syntax" of the inductive and recursive types we present. We also borrow notions from regular expressions and program calculation to describe some of the types. We present in Table 3.2 our notation.

Representation	Stands for
$A^* = 1 \mid A\ A^*$	0 or more A.
$A^+ = A\ A^*$	At least one A.
Maybe $A = 1 \mid A$	Nothing or A.
$A \times B$	Tupling of types A and B.
$A + B$	Choice operator. It is **either** A or B.
$A \to B$	Mapping from type A to B.

Table 3.2: Notation used for type descriptions.

Types and Environment In Type Signature 1 we define the notion of environment used in our semantics. This environment represents the information available at any given moment to every inference rule which has an environment present in its definition. As we can see, the environment is simply a tuple that aggregates information on representation types, variables, procedures and functions. Representation types in mSPARK are types defined by the programmer.

Type Signature 1 Definition of Environment.

$$\text{Environment} = \text{ Environment } \mathit{TMap} \times \mathit{VMap} \times \mathit{PMap} \times \mathit{FMap} \mid \text{EnvError Error} \tag{3.1}$$

$$\mathit{TMap} = ID_t^* \times \mathcal{RT} \tag{3.2}$$

$$\mathit{VMap} = ID_v^* \times \mathcal{V} \tag{3.3}$$

$$\mathit{PMap} = ID_p^* \times \mathcal{P} \tag{3.4}$$

$$\mathit{FMap} = ID_f^* \times \mathcal{F} \tag{3.5}$$

Type Signature 2 defines the type $\mathcal{V}$. It maps variable identifiers (ID_v) to one of the following: a) Nil, which means that the variable does not exist or b) to a pair where the first component is the identifier of the type (ID_v) (that represents the type) of the variable, and the second component (of type $\mathcal{T}$) which is the value that the variable holds.

Type Signature 2 Type of the $\mathcal{V}$ function and inductive definition of atomic datatypes ($\mathcal{T}$).

$$\mathcal{V} = ID_v \rightarrow \text{Maybe } ID_t \times \mathcal{T} \tag{3.6}$$

$$\mathcal{T} = \tag{3.7}$$

$$\mathit{TError} \mid \mathit{Empty} \mid \mathit{TBool\ Boolean} \mid \mathit{TChar\ Char} \mid \mathit{TInt\ Int} \mid$$
$$\mathit{TReal\ Real} \mid \mathit{TString\ String} \mid \mathit{TArray}\ (\mathcal{T}^+ \ \rightarrow\ \mathcal{T})$$

The inductive definition of $\mathcal{T}$ corresponds to the atomic types in our language. It should be noted that we allow the distinction of having or not having a value, by allowing the Empty constructor. We also have the Error constructor for the representation type because, as we will see, this is important for the treatment of the evaluation functions for expressions as total functions.

Some textbooks define the available types using a meta-variable τ which usually ranges over only int and bool. Our $\mathcal{T}$ needs more primitive types (and constructors for those types) because of the richness of the language.

While most types are fairly intuitive, TArray deserves further explanation. TArray maps a non-empty list of $\mathcal{T}$ to $\mathcal{T}$. This represents multi-dimensional arrays as a function that maps its "coordinates" to a value. As an example, we could have $[\mathit{TInt}\ 1,\ \mathit{TChar}\ 'a',\ \mathit{TBool}\ \mathit{False}] \mapsto \mathit{TString}$ "something", which would be a perfectly valid type according to Ada and SPARK, for example.

In Type Signature 3 we define the type $\mathcal{P}$, corresponding to functions that represent procedures by mapping procedure identifiers (ID_t) to tuples that consist of a list of parameters, the procedure's local state (which may superimpose over the global state) and the statements

Type Signature 3 Type of the $\mathcal{P}$ function.

$$\mathcal{P} = ID_p \rightarrow \text{Maybe } \mathit{Contract} \times \mathit{Parameters}_p \times \text{Local}_p \times S \tag{3.8}$$
$$\mathit{Parameters}_p = ((\mathit{Mode} \times ID_v)^* \times \mathcal{V}) \tag{3.9}$$
$$\text{Mode} = \{\mathit{In}, \mathit{Out}, \mathit{In\_Out}\} \tag{3.10}$$
$$\text{Local}_p = \mathit{VMap} \times \mathit{PMap} \times \mathit{FMap} \tag{3.11}$$

of the body.

The parameters are a type renaming for a tuple which has a list of pairs of variable identifiers and their associated modes (in, out, inout), and a variable function. The local state of procedures may include local variables (*VMap*) as well as nested procedures and functions (subprograms).

Type Signature 4 Type of the $\mathcal{F}$ function.

$$\mathcal{F} = ID_f \rightarrow \text{Maybe } \mathit{Contract} \times \mathit{Parameters}_f \times ID_t \times \text{Local}_f \times S \times E \tag{3.12}$$
$$\mathit{Parameters}_f = \mathit{VMap} \tag{3.13}$$
$$\text{Local}_f = \mathit{VMap} \times \mathit{FMap} \tag{3.14}$$

Type Signature 4 is analogous to Type Signature 3, now regarding functions. It maps each function identifier (ID_f) to a tuple consisting of parameters, the id of the return representation type of the function, its local state, and the return expression. The parameters of a function are simply a *VMap* and a *FMap*. It is a design choice not to allow procedures inside functions.

Type Signature 5 Type of $\mathcal{RT}$ and the inductive definition of $\mathcal{R}$.

$$\mathcal{RT} = ID_t \rightarrow \text{Maybe } (ID_t^+ + \text{Maybe } ID_t \times \mathcal{R}) \tag{3.15}$$
$$\mathcal{R} \ = \ \mathit{RInt}\ \mathit{Int}\ \mathit{Int} \mid \mathit{RChar}\ \mathit{Char}\ \mathit{Char} \mid \mathit{REnum}\ ID_t\ ID_t \mid \mathit{RArray}\ \mathcal{R}^+\ \mathcal{R} \tag{3.16}$$

Type Signature 5 is perhaps one of the most important. $\mathcal{RT}$ corresponds to functions that represent the range types (hence the use of the name "representation types"), a major focus of our semantics. $\mathcal{R}$ is the inductive definition of what types of ranges we allow in our language.

$\mathcal{RT}$ maps type identifiers (ID_t) to a choice: a) a non-empty list of identifiers for constructors of that type or b) a pair indicating whether this type has a base type or not (Maybe constructor) and its range ($\mathcal{R}$).

As we can see in the $\mathcal{R}$ definition, we allow ranges to be pairs of ints, pairs of characters, pairs of constructors from enumeration types, and even n-dimensional arrays parametrized by a non-empty list of $\mathcal{R}$, which defines the range of each dimension of the array, and a $\mathcal{R}$ which gives the range of the values contained in the array.

Additional definitions To define our operational semantics, we will also need some additional definitions. Table 3.3 presents the types of expressions that are allowed in our language, in an informal way, providing only their intuition. Table 3.4 presents the evaluation function for each expression type that, given an expression and an environment, yields a value (unless the environment is an **error** or the expression as a safety error (division by 0, for instance), which in that case the evaluation functions yield an **error_type**).

Name	Stands for
$BExpr$	Boolean expressions.
$AExpr$	Arithmetic expressions (may use variable values and calls to functions).
$CAExpr$	Constant arithmetic expressions. May only use atomic values, such as 0, False or T'First.
$Expr$	Any type of expression.

Table 3.3: mSPARK Expressions.

Meta-variables To aid us in the following definitions, we will use the following meta-variables that will refer to instances of these constructs at a meta-level. Notice that the names on the right side of :: have a correspondence in the abstract syntax that was presented in Figure 3.6, such as Global and Assert, while others were presented in the type signatures or in previous tables.

- env :: Environment
- $\{t, bt\}$ :: ID_t
- p :: ID_p
- f :: ID_f
- $\{x, a, v\}$ :: ID_v, with x being a regular variable, a an array and v ranging over variables and arrays.
- e :: $Expr$
- b, be :: $BExpr$
- ae :: $AExpr$

- $ce :: CAExpr$
- $k :: \mathcal{T}$
- $r :: \mathcal{R}$
- $\{\theta, \psi\} :: Assert$
- $\mathcal{G} :: Global$

Additionally, we will use S to represent a statement of the language[2] (such as x := e).

Name	Evaluation function of
$be \Downarrow_{\text{BExpr}(env)} t$	Boolean expressions. Given a *be* and an *env* it yields a value t.
$ae \Downarrow_{\text{AExpr}(env)} t$	Arithmetic expressions. Given an *ae* and an *env* it yields a value t.
$cae \Downarrow_{\text{CAExpr}(env)} t$	Constant arithmetic expressions. Given a *cae* and an *env* it yields a value t. May need the environment to access type information (such as T'First or enumeration constructors).
$e \Downarrow_{\text{Expr}(env)} t$	Expressions. Evaluates any type of expression at a given environment *env*.
$D_r \Downarrow_{\text{Dr}} r$	Declaration of ranges. Evaluates range declarations into ranges.

Table 3.4: Evaluation Functions.

Table 3.4 describes the evaluation functions used for expressions and range declarations. Notice that declaring a range does not need an environment.

The transitions presented in Table 3.5 are used to separate the evaluation of several type of constructs, from executable statements to several declaration constructs. This is needed to differentiate how each construct should be handled, by using a different transition.

The functions in Table 3.6 are used in the inference rules of our operational semantics. Before showing the rules themselves, we give some intuition on the auxiliary functions that they use. This also serves as a "cheat sheet" when reading the operational semantics, if there is any doubt to what each function is supposed to do.

Regarding notation, we use the [e/x] notation for replacement, with the usual interpretation (x is replaced by e). The notation $a([k_1 \ldots k_n] \rhd k)$ is used to represent a change to array a, at index $(k_1 \ldots k_n)$, to the value k; this yields a new array.

[2]As we will see, if we want several statements, we need to use a composition statement

Name	Transition
$(S,\ env) \rightsquigarrow env'$	Transitions from *env* to *env'* after executing the statement S.
$(dt,\ env) \rightsquigarrow_{\mathrm{Dt}} env'$	Transitions from *env* to *env'*. Used for type declarations.
$(dv,\ env) \rightsquigarrow_{\mathrm{Dv}} env'$	Transitions from *env* to *env'*. Used for variable declarations.
$(dsp,\ env) \rightsquigarrow_{\mathrm{Df}} env'$	Transitions from *env* to *env'*. Used for subprogram declarations, where the subprogram is a function.
$(dsp,\ env) \rightsquigarrow_{\mathrm{Dsp}} env'$	Transitions from *env* to *env'*. Used for subprogram declarations.

Table 3.5: Transition Relations.

3.2.4 Natural Semantics of mSPARK

In this section we present the formal semantics of the mSPARK programming language, defined in the Natural Semantics style.

Inference Rules of the Natural Semantics Recall that the evaluation of an expression in an error state, as the evaluation of an invalid expression, yields a constant **error_type**. The boolean operator ::, when trying to check if the type of a constant fits a certain range yields $\perp$ if: a) the constant is of value **error_type** or b) the constant is out of the range of the representation type. Also, $\not\colon$ is used as an abbreviation of $\neg(k :: \mathcal{T})$. The meta-value **error** is an identifier that encapsulates all error constants.

The Inference Rules 1, 2 and 3 introduce the executable statements of our natural semantics. These executable statements assume that they have a meaningful environment that allows them to be executed and to check range constraints on the types. In rules 4, 5, 6 and 7 we explore how such an environment can be created. Rules 4 and 5 deal with type and variable declarations and rules 6 and 7 deal with subprogram (functions and procedures) declarations.

In Inference Rules 1 we present the basic commands that are usually available in While-like programming languages, with the addition of type checking for expressions, handling error states and allowing for multi-dimensional arrays.

In Inference Rules 2 we present the basic control structures that are usually present in While-like programming languages. We also are careful with contemplating the cases where an evaluation error might/should occur.

Do recall the (Error) rule in Inference Rules 1. This is of special importance for the inference rule of the While loop where the body is executed while the condition of the loop is true and

Function	Type	Intuition
$variable\_type$	Environment $\times ID_v \to ID_t$	Gets the type of the variable, in the scope of the environment.
$type\_array\_ind$	Environment $\times ID_v \to ID_t^+$	Gets the list of types of the indexes of an n-dimensional array.
$type\_array\_val$	Environment $\times ID_v \to ID_t$	Gets the type of the values of an array.
$valid\_range$	Environment$\times ID_t \times \mathcal{R} \to Bool$	Determines if the range is valid for the type.
$addAssignVar$	Environment $\times ID_t^+ \times ID_t \times \mathcal{T} \to$ Environment	Adds a non-empty list of variables of the given type and assigns them the given value.
$remove\_var$	Environment $\times ID_v \to$ Environment	Removes the variable from the environment.
$addRangeType$	Environment $\times ID_v \times \mathcal{R} \to$ Environment	Adds a type to the environment.
$addRangeSubType$	Environment $\times ID_v \times \mathcal{R} \to$ Environment	Adds a subtype to the environment, with a given basetype.
$addFunctionAndVar$	Environment $\times ID_f \times ID_t \to$ Environment	Adds a function name (and a return variable for the function) to the environment.
$reduceState$	Environment $\to$ Environment	Reduces the state to a minimum (maintains only the types).
upd_f	$ID_f \times$ **Assert** $\times$ **Assert** $\times S \times Params_f \times Env \times Env \to$ Environment	Enriches the function with its local environment and body.
$addProcedure$	$ID_p \times$ Environment $\to$ Environment	Adds procedure p to the environment.
upd_p	$ID_p \times$ **Global** $\times$ **Assert** $\times$ **Assert** $\times S \times Env \to$ Environment	Enriches the procedure with its local environment and body.
$getEnvP$	Environment $\times ID_p \to$ Environment	Enriches the environment with the definition of the procedure.
$getBodyProcedure$	Environment $\times ID_p \to S$	Retrieves the body of the procedure.
$restoreLocalVariables$	Environment $\times$ Environment $\times ID_p \to$ Environment	Restores the original values of variables in the environment that may have been overwritten by the procedure.
$getEnvF$	Environment $\times ID_f \to$ Environment	Enriches the environment with the definition of the function.
$getBodyFunction$	Environment $\times ID_f \to S$	Retrieves the body of the function.
$getReturnType$	Environment $\times ID_f \to ID_t$	Retrieves the return type of the function.
$getReturnExpression$	Environment $\times ID_f \to E$	Retrieves the return expression of the function.
$getParamList$	Environment $\times ID_f \to VMap$	Retrieves the parameter list of the function.

Table 3.6: Auxiliary Functions for the Natural Semantics.

Inference Rules 1 Basic statement rules, including array assignment.

$$\frac{}{(\mathbf{null},\ env) \rightsquigarrow env} \qquad \text{(Null)}$$

$$\frac{}{(S,\ \mathbf{error}) \rightsquigarrow \mathbf{error}} \qquad \text{(Error)}$$

$$\frac{(S_1,\ env) \rightsquigarrow env' \qquad (S_2,\ env') \rightsquigarrow env''}{(S_1; S_2,\ env) \rightsquigarrow env''} \qquad \text{(Composition)}$$

$$\frac{e\ \Downarrow_{\mathrm{Expr}(env)}\ k \qquad k :: variable\_type(env, x)}{(x := e,\ env) \rightsquigarrow env[k/x]} \qquad \text{(AssignVar)}$$

$$\frac{e\ \Downarrow_{\mathrm{Expr}(env)}\ k \qquad k \not:: variable\_type(x, env)}{(x := e,\ env) \rightsquigarrow \mathbf{constraint\_error}} \qquad (\mathrm{AssignVar}_{typeError})$$

$$\frac{\begin{array}{c}[e_1, \ldots, e_n]\ \Downarrow_{\mathrm{Expr}(env)}\ [k_1, \ldots, k_n] \qquad e\ \Downarrow_{\mathrm{Expr}(env)}\ k \\ [k_1, \ldots, k_n] :: type\_array\_ind(env, a) \qquad k :: type\_array\_val(env, a)\end{array}}{(a(e_1, \ldots, e_n) := e,\ env) \rightsquigarrow env[a([k_1 \ldots k_n] \rhd k)/a]} \ \text{(AssignArr)}$$

$$\frac{[e_1, \ldots, e_n]\ \Downarrow_{\mathrm{Expr}(env)}\ [k_1, \ldots, k_n] \qquad [k_1, \ldots, k_n] \not:: type\_array\_ind(env, a)}{(a(e_1, \ldots, e_n) := e,\ env) \rightsquigarrow \mathbf{constraint\_error}}$$
$$(\mathrm{AssignArr}_{index\_error})$$

$$\frac{\begin{array}{c}[e_1, \ldots, e_n]\ \Downarrow_{\mathrm{Expr}(env)}\ [k_1, \ldots, k_n] \qquad e\ \Downarrow_{\mathrm{Expr}(env)}\ k \\ [k_1, \ldots, k_n] :: type\_array\_ind(env, a) \qquad k \not:: type\_array\_val(env, a)\end{array}}{(a(e_1, \ldots, e_n) := e,\ env) \rightsquigarrow \mathbf{constraint\_error}}$$
$$(\mathrm{AssignArr}_{value\_error})$$

Inference Rules 2 Basic control structures.

$$\frac{b \Downarrow_{\text{BExpr}(env)} \textbf{\textit{true}} \qquad (S_1,\ env) \rightsquigarrow env'}{(\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end if}\ ,\ env) \rightsquigarrow env'} \qquad (\text{IfThenElse}_1)$$

$$\frac{b \Downarrow_{\text{BExpr}(env)} \textbf{\textit{false}} \qquad (S_2,\ env) \rightsquigarrow env'}{(\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end if}\ ,\ env) \rightsquigarrow env'} \qquad (\text{IfThenElse}_2)$$

$$\frac{b \Downarrow_{\text{BExpr}(env)} \textbf{error}}{(\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end if}\ ,\ env) \rightsquigarrow \textbf{error}} \qquad (\text{IfThenElse}_{\textbf{error}})$$

$$\frac{b \Downarrow_{\text{Expr}(env)} \textbf{\textit{true}} \qquad (S,\ env) \rightsquigarrow env' \qquad (\textbf{while}\ \theta\ b\ \ \textbf{loop}\ S\ \textbf{end loop}\ ,\ env') \rightsquigarrow env''}{(\textbf{while}\ \theta\ b\ \ \textbf{loop}\ S\ \textbf{end loop}\ ,\ env) \rightsquigarrow env''} \qquad (\text{WhileLoop}_1)$$

$$\frac{b \Downarrow_{\text{BExpr}(env)} \textbf{\textit{false}}}{(\textbf{while}\ \theta\ b\ \ \textbf{loop}\ S\ \textbf{end loop}\ ,\ env) \rightsquigarrow env} \qquad (\text{WhileLoop}_2)$$

$$\frac{b \Downarrow_{\text{BExpr}(env)} \textbf{error}}{(\textbf{while}\ \theta\ b\ \ \textbf{loop}\ S\ \textbf{end loop}\ ,\ env) \rightsquigarrow \textbf{error}} \qquad (\text{WhileLoop}_{error1})$$

stops when b is either false or an error or the evaluation of the statements in the body yields an error. The (Error) rule makes possible that this error is propagated automatically without having to define specific rules for intermediate error states.

Also note that while the while command has an associated assertion, e.g. loop invariant, this assertion does not interfere with our operational semantics because we do not execute assertions. Eventually there could exist an operational description for the execution of assertions at run-time, although this was not explored in our work. We refer to this again in 5.

We present in Inference Rules 3 rewrite rules for other more complex control structures. These control structures can be rewritten in a pretty straightforward fashion using rules from 1 and 2. The exception is the for loop, where we, following the Ada convention, have to create a variable of type t and release it after the loop execution. This also implies that every type that can be used in a for loop must have a successor function. As a simplification, we do not allow for loops with decreasing steps.

In Inference Rules 4, Dr stands for "Declaration of Ranges" and r is the particular range that the declaration yields after being evaluated by $\Downarrow_{\text{Dr}}$.

The evaluation of a type declaration expression uses a new transition denoted by $\rightsquigarrow_{\text{Dt}}$. While in previous rules we used the transition $\rightsquigarrow$ without indexing it, it becomes necessary

Inference Rules 3 Extended control structures.

$$\frac{(\textbf{if } b \textbf{ then } S \textbf{ else null end if}\ ,\ env) \rightsquigarrow env'}{(\textbf{if } b \textbf{ then } S \textbf{ end if}\ ,\ env) \rightsquigarrow env'} \quad \text{(If)}$$

$$\frac{(\textbf{if } b_1 \textbf{ then } S_1 \textbf{ else } (\textbf{if } b_2 \textbf{ then } S_2 \textbf{ end if}\ ) \textbf{ end if}\ ,\ env) \rightsquigarrow env'}{(\textbf{if } b_1 \textbf{ then } S_1 \textbf{ elsif } b_2 \textbf{ then } S_2 \textbf{ end if}\ ,\ env) \rightsquigarrow env'} \quad \text{(IfElsIf)}$$

$$\frac{(\textbf{if } b_1 \textbf{ then } S_1 \textbf{ else } (\textbf{if } b_2 \textbf{ then } S_2 \textbf{ else } S_f \textbf{ end if}\ ) \textbf{ end if}\ ,\ env) \rightsquigarrow env'}{(\textbf{if } b_1 \textbf{ then } S_1 \textbf{ elsif } b_2 \textbf{ then } S_2 \textbf{ else } S_f \textbf{ end if}\ ,\ env) \rightsquigarrow env'} \quad \text{(IfElsIfElse)}$$

$$\frac{\begin{array}{c}(\textbf{if } b_0 \textbf{ then } S_0 \textbf{ else} \\ (\textbf{if } b_1 \textbf{ then } S_1 \textbf{ elsif } b_2 \textbf{ then } S2\ \ldots\ \textbf{elsif } b_n \textbf{ then } S_n \textbf{ end if}\ ) \\ \textbf{end if}\ ,\ env) \rightsquigarrow env'\end{array}}{(\textbf{if } b_0 \textbf{ then } S_0 \textbf{ elsif } b_1 \textbf{ then } S_1 \textbf{ elsif } b_2 \textbf{ then } S2\ \ldots\ \textbf{elsif } b_n \textbf{ then } S_n \textbf{ end if}\ ,\ env) \rightsquigarrow env'} \quad \text{(IfListElsif)}$$

$$\frac{\begin{array}{c}(\textbf{if } b_0 \textbf{ then } S_0 \textbf{ else} \\ (\textbf{if } b_1 \textbf{ then } S_1 \textbf{ elsif } b_2 \textbf{ then } S_2\ \ldots\ \textbf{elsif } b_n \textbf{ then } S_n \textbf{ else } S_f \textbf{ end if}\ ) \\ \textbf{end if}\ ,\ env) \rightsquigarrow env'\end{array}}{\begin{array}{c}(\textbf{if } b_0 \textbf{ then } S_0 \\ \textbf{elsif } b_1 \textbf{ then } S_1 \textbf{ elsif } b_2 \textbf{ then } S_2 \ldots\ \textbf{elsif } b_n \textbf{ then } S_n \\ \textbf{else } S_f \textbf{ end if}\ ,\ env) \rightsquigarrow env'\end{array}} \quad \text{(IfListElsifElse)}$$

$$\frac{(S,\ env) \rightsquigarrow env' \qquad (\textbf{while not } b\ \ \textbf{loop } S \textbf{ end loop}\ ,\ env') \rightsquigarrow env''}{(\ \textbf{loop } S \textbf{ exit when } b \textbf{ end loop}\ ,\ env) \rightsquigarrow env''} \quad \text{(LoopExitWhen}_1\text{)}$$

$$\frac{(\textbf{while } \theta \textbf{ not } b\ \ \textbf{loop } S \textbf{ end loop}\ ,\ env) \rightsquigarrow env'}{(\ \textbf{loop}\ \ \theta \textbf{ exit when } b\ S \textbf{ end loop}\ ,\ env) \rightsquigarrow env'} \quad \text{(LoopExitWhen}_2\text{)}$$

$$\frac{\begin{array}{c}D_r\ \Downarrow_{\text{Dr}}\ r \qquad valid\_range(env, t, r) \\ env' = addAssignVar(env, [x], t, r\text{'First}) \\ (\textbf{while } \theta\ (x <= r\text{'Last})\ \ \textbf{loop } S\ ;\ x := succ(env, x) \textbf{ end loop}\ ,\ env') \rightsquigarrow env'' \\ env''' = removeVar(env'', x)\end{array}}{(\textbf{for } x \textbf{ in } t \textbf{ range } D_r \textbf{ loop } \theta\ S \textbf{ end loop}\ ,\ env) \rightsquigarrow env'''} \quad \text{(ForLoop)}$$

$$\frac{D_r\ \Downarrow_{\text{Dr}}\ r \qquad \neg valid\_range(env, t, r)}{(\textbf{for } x \textbf{ in } t \textbf{ range } r \textbf{ loop } \theta\ S \textbf{ end loop}\ ,\ env) \rightsquigarrow \textbf{constraint\_error}} \quad \text{(ForLoop}_{\textbf{error}}\text{)}$$

Inference Rules 4 Type declarations.

$$\frac{D_r \ \Downarrow_{\mathrm{Dr}}\ r \qquad (D_t,\ env') \rightsquigarrow_{\mathrm{Dt}} env''}{(\textbf{type}\ t\ \textbf{is}\ D_r;\ D_t,\ env) \rightsquigarrow_{\mathrm{Dt}} env''} \qquad (\text{Type}_{decl})$$

$$\textbf{where}\ env' = addRangeType(env, t, r)$$

$$\frac{D_r \ \Downarrow_{\mathrm{Dr}}\ r \qquad (D_t,\ env') \rightsquigarrow_{\mathrm{Dt}} env''}{(\textbf{subtype}\ t\ \textbf{is}\ bt\ D_r;\ D_t,\ env) \rightsquigarrow_{\mathrm{Dt}} env''} \qquad (\text{Subtype}_{decl})$$

$$\textbf{where}\ env' = addRangeSubtype(env, t, bt, r)$$

$$\frac{}{(\epsilon,\ env) \rightsquigarrow_{\mathrm{Dt}} env} \qquad (\text{NoType}_{decl})$$

to do this to clearly identify and separate the several different transitions that we need in our semantics.

We allow the definition of types and subtypes in our semantics. A type is a base definition that relies on its ranges to be fully defined, that is, if we define a range that its first and last elements are integers then that type will have the basic operations of integers; the same applies for all other basic types, such as enumerations and characters.

Subtypes are types which have an underlying base type. This means that this new type must be defined within the ranges of its base type; if this is not the case, then the function addRangeSubtype should yield an environment with an error value. The subtype feature is quite useful when developing SPARK programs, in particular for discharging verification conditions related to arrays and safety.

In Inference Rules 5 we present the rules for variable declarations and introduce the $(D_v,\ env) \rightsquigarrow_{\mathrm{Dv}} env$ transition for variable declaration evaluation. We allow for several variable declarations with the same type and assigning/initializing them to a value. This value must be a constant expression such as "$2 + 3$" or "Integer'First"; expressions that can be evaluated statically. This initialization value has to respect the type t of the variable.

Inference Rules 6 formalizes the rules for function declaration. Functions are meant to be pure and so they can not alter the state (or should not, since we are not enforcing this on this version of the static semantics).

Functions can read all visible variables in the scope, as well as declaring new local variables. Local variables override variables that were already in the scope with the same identifier. This override is only enforced after the function in invoked.

Inference Rules 5 Variable declarations.

$$\frac{(D_v,\ env') \rightsquigarrow_{\mathrm{Dv}} env''}{(x_1, x_2, \ldots, x_n\ :\ t;\ D_v,\ env) \rightsquigarrow_{\mathrm{Dv}} env''} \quad (\mathrm{var}_{decl1})$$
$$\textbf{where } env' = addVarAssign(env, [x_1, x_2, \ldots, x_n], t, Empty)$$

$$\frac{ce\ \Downarrow_{\mathrm{CAExpr}(env)}\ k \qquad k :: t \qquad (D_v,\ env') \rightsquigarrow_{\mathrm{Dv}} env''}{(x_1, x_2, \ldots, x_n\ t := ce;\ D_v,\ env) \rightsquigarrow_{\mathrm{Dv}} env''} \quad (\mathrm{var}_{decl2})$$
$$\textbf{where } env' = addVarAssign(env, [x_1, x_2, \ldots, x_n], t, k)$$

$$\frac{ce\ \Downarrow_{\mathrm{CAExpr}(env)}\ k \qquad k \not:: t}{(x_1, x_2, \ldots, x_n\ :\ t := ce;\ D_v,\ env) \rightsquigarrow_{\mathrm{Dv}} \textbf{type\_error}} \quad (\mathrm{var}_{decl\_type\_error})$$

$$\frac{}{(\epsilon,\ env) \rightsquigarrow_{\mathrm{Dv}} env} \quad (\mathrm{noVar}_{\mathrm{decl}})$$

In these rules we purposely reduce the state that is visible to the evaluation relation so that the declaration of the function only knows its variables. This reduces the amount of information we need to keep track of when invoking a function.

Functions can also have nested functions. These nested functions are local to the function in which they have been declared and their identifiers must be unique; overriding rules are not applied to function names.

Also, functions are only allowed to have parameters without modes. In Ada and SPARK (and in mSPARK) this means that the parameters can only be read. We can not alter their values as, for example, it is possible to do in C.

Another important feature is that the return value of a function is always given in the last statement of a function; this allows for simpler reasoning about the return of a function. Also, because functions are pure, the only way to communicate its value to the outside world is if we store the value of the evaluation of the return expression in a variable of the outer scope. For this reason, when we create the function in the environment we also associate a variable to it.

We also enforce the existence of contracts in our functions by not allowing a contract to be empty (although the contract may be something trivial such as the assertion **true**).

Inference Rules 7 handles the declaration of procedures. We only handle parameterless procedures in our semantics definition. This is because procedures using parameters with arbitrary modes of variable passing (in, out and inout) are harder to reason about, particularly at the level of axiomatic semantics; this is the main reason for not including procedures with parameters.

Because procedures do not have parameters, they can only communicate to the outside

Inference Rules 6 Function declarations.

$$\frac{(D_v,\ env') \rightsquigarrow_{\mathrm{Dv}} env'' \qquad (D_f,\ env'') \rightsquigarrow_{\mathrm{Df}} env''' \qquad (D'_f,\ env^{(4)}) \rightsquigarrow_{\mathrm{Df}} env^{(5)}}{(\textbf{function}\ f\ \textbf{return}\ t\ \textbf{is}\ \theta\ \psi\ D_v\ D_f\ \textbf{begin}\ S;\ \textbf{return}\ e\ \textbf{end}\ f;\ D'_f,\ env) \rightsquigarrow env^{(5)}}$$

(ParameterlessFunction$_{decl}$)

where
$env_0 = addFunctionAndVar(env, f, t)$
$env' = reduceState(env)$
$env^{(4)} = upd_f(f, \theta, \psi, S, e, [], env''', env_0)$

$$\frac{(D_v,\ env') \rightsquigarrow_{\mathrm{Dv}} env'' \qquad (D_f,\ env'') \rightsquigarrow_{\mathrm{Df}} env''' \qquad (D'_f,\ env^{(4)}) \rightsquigarrow_{\mathrm{Df}} env^{(5)}}{(\textbf{function}\ f(p_1 : t_1, ..., p_n : t_n)\ \textbf{return}\ t\ \textbf{is}\ \theta\ \psi\ D_v\ D_f\ \textbf{begin}\ S;\ \textbf{return}\ e\ \textbf{end}\ f; D'_f,\ env) \rightsquigarrow env^{(5)}}$$

(Function$_{decl}$)

where
$env_0 = addFunctionAndVar(env, f, t)$
$env' = reduceState(env)$
$env^{(4)} = upd_f(f, \theta, \psi, S, e, f, [p1 : t_1, ..., p_n : t_n], env''')$

$$\frac{}{(\epsilon,\ env) \rightsquigarrow_{\mathrm{Df}} env}$$

(noFunction$_{decl}$)

Inference Rules 7 Procedure and subprogram declarations.

$$\frac{(D_v,\ env') \rightsquigarrow_{\mathrm{Dv}} env'' \qquad (D_{sp_1},\ env'') \rightsquigarrow_{\mathrm{Dsp}} env''' \qquad (D_{sp_2},\ env^{(4)}) \rightsquigarrow_{\mathrm{Dsp}} env^{(5)}}{(\textbf{procedure}\ p\ \textbf{is}\ \ \mathcal{G}\ \theta\ \psi\ D_v\ D_{sp_1}\ \textbf{begin}\ S\ \textbf{end}\ p;\ D_{sp_2},\ env) \rightsquigarrow env^{(5)}}$$

(ParamaterlessProcedure$_{decl}$)

where
$env' = addProcedure(p, env)$
$env^{(4)} = upd_p(p,\ \mathcal{G}\ , \theta, \psi, S, env''')$

$$\frac{(\textbf{function}\ \ldots,\ env) \rightsquigarrow_{\mathrm{Df}} env' \qquad (D_{sp},\ env') \rightsquigarrow_{\mathrm{Dsp}} env''}{(\textbf{function}\ \ldots; D_{sp},\ env) \rightsquigarrow_{\mathrm{Dsp}} env''}\ (\text{Function}_{decl\_sp})$$

$$\frac{}{(\epsilon,\ env) \rightsquigarrow_{\mathrm{Dsp}} env}\qquad (\text{noSubprogram}_{decl})$$

world by writing to non-local variables visible in that scope. This means not only global variables but variables that are in outer blocks. As an example, if we have a nested procedure inside a nested procedure, the innermost procedure can access the local variables of the outermost procedure.

In these rules we also find a very special transition. Procedures use a $\rightsquigarrow_{D_{\mathrm{sp}}}$ transition for their declaration evaluation. As with functions, procedures can have nested procedures and nested functions but when a function is nested inside a procedure, we can not evaluate that declaration with the same $\rightsquigarrow_{D_{\mathrm{sp}}}$ because that would mean that functions could now have nested procedures that could alter the state of the program thus making the functions not pure. To avoid this, whenever the $\rightsquigarrow_{D_{\mathrm{sp}}}$ transition encounters a function, it changes back to the $\rightsquigarrow_{D_{\mathrm{f}}}$ transition.

Procedure invocation is pretty straightforward. We just enrich our initial environment with the local variables and nested subprograms and then we execute the procedure body. It is important that after the execution of a procedure we restore the value of non-local variables that our local variables may have overridden.

For parameterless functions this is also a quite straightforward process, much similar to procedures. Since functions do not alter state, we just add to the initial state the return value of the function and discard any possible intermediate states that the invocation of the function may have generated.

It is also clear that we need to check if the evaluation of the return of the function yields

Inference Rules 8 Procedure invocation.

$$\frac{(S,\ env') \rightsquigarrow env''}{(p,\ env) \rightsquigarrow env'''} \qquad (\text{call}_{proc})$$

where
$env' = getEnvP(env, p)$
$S = getBodyProcedure(env, p)$
$env''' = restoreLocalVariables(env, env'', p)$

Inference Rules 9 Function invocation.

$$\frac{(S,\ env') \rightsquigarrow env'' \qquad exp \Downarrow_{\text{Expr}(env'')} k \qquad k :: f_t}{env \models f \Rightarrow env[k/f]} \qquad (\text{call}_{function})$$

where
$env' = getEnvF(env, f)$
$S = getBodyFunction(env, f)$
$exp = getReturnExpression(env, f)$
$f_t = getReturnType(env, f)$

$$\frac{\begin{array}{c} e_1 \Downarrow_{\text{Expr}(env')} k_1 \ldots e_n \Downarrow_{\text{Expr}(env')} k_n \\ k_1 :: same\_type(env, p_1, V) \ldots k_n :: same\_type(env, p_n, V) \\ env_1 = upd_{param}(env, p_1, k_1, V) \ \ldots \ env_n = upd_{param}(env_{n-1}, p_n, k_n, V) \\ (S,\ env_n) \rightsquigarrow env'' \qquad exp \Downarrow_{\text{Expr}(env'')} k \qquad k :: f_t \end{array}}{(f(e_1, \ldots, e_2),\ env) \rightsquigarrow env[k/f]}$$

$(\text{call}_{function\_with\_parameters})$

where
$env' = getEnvF(env, f)$
$S = getBodyFunction(env, f)$
$exp = getReturnExpression(env, f)$
$([p_1 \ \ldots \ p_n], V) = getParamList(env, f)$
$f_t = getReturnType(env, f)$

a value that is typable by the return type of function f.

As for functions with parameters, we need to evaluate expressions passed as parameters and check if they type correctly regarding the types that have been declared for parameters. After assigning the values that the expressions yielded to the corresponding parameters, a function with parameters behaves exactly the same as a parameterless function.

Inference Rules 10 mSPARK entry point.

$$\frac{\begin{array}{c}(D_t,\ \mathit{Initial\_State}) \leadsto_{\mathrm{Dt}} env' \\ (D_v,\ env') \leadsto_{\mathrm{Dv}} env'' \qquad (D_{sp},\ env'') \leadsto_{\mathrm{Dsp}} env''' \qquad (S,\ env''') \leadsto env^{(4)}\end{array}}{(\textbf{types}\ D_t\ \textbf{variables}\ D_v\ \textbf{subprograms}\ D_{sp}\ \textbf{execute}\ S\ \textbf{end},\ \mathit{Initial\_State}) \leadsto env^{(4)}}\ \text{(EntryPoint)}$$

In Inference Rules 10 we define what is a mSPARK program and its entry point.

Since our language has not, as of yet, support for packages, we manufactured an entry point. Although this deviates from SPARK, we took this approach so that it would be easy to create programs using all the things that we have defined, without having the need to introduce packages.

We evaluate the declarations in the most useful order (types, variables and subprograms) and then we execute the statements that are in the execute block.

A particularity of this rule is that it has a clearly defined environment that is named Initial_Environment. This initial environment can be anything, from an empty state to, hypothetically, other programs/packages with their own types and declarations. To support this, our notion of environment would need to evolve but this seems something that is feasible. We would also need to change rules related to variables and subprogram invocation, so that they would take into account these multiple programs/packages.

3.3 Hibana-chan: Type Checking and Animating the Operational Semantics of mSPARK

An operational semantics specified in the natural semantics style can easily be implemented as an interpreter for a language.

It is customary to represent the language constructs by inductive type definitions in a functional programming language and to define its evaluation functions over these types. This is usually what is called “language embedding” because we are defining a specific programming language inside another programming language and using the features of this host programming language to ease the implementation of the interpreter.

Haskell [Jon03] was a natural choice for us because of our large experience with the language but also because of its type mechanism.

While we were defining the operational semantics, we did not have any feedback about what we were writing; they were just mathematical symbols on "paper". By representing the constructs and rules in Haskell, without even having to implement the functions, it gave us feedback about the typing of our rules. In Code 2 we show how we define the inference rules.

Code 2 Haskell Types for Statements.

```
data Statements = Null |
          Composition Statements Statements |
          AssignVar ID_v E |
          AssignArr ID_v [E] E |
          IfThenElse BE Statements Statements |
          While Assert BE Statements |
          If BE Statements |
          IfElsIf BE Statements BE Statements |
          IfElsIfElse BE Statements BE Statements Statements |
          IfListElsIf BE Statements [(BE, Statements)] |
          IfListElsIfElse BE Statements [(BE, Statements)] Statements |
          InfiniteLoop Statements |
          LoopExitWhen1 Statements BE |
          LoopExitWhen2 Assert BE Statements |
          Declaration_Var D_v |
          Declaration_Function D_f |
          Declaration_Subprogram D_sp |
          Declaration_Type D_t |
          CallPF ID_f |
          CallF ID_f [E] |
          CallPP ID_p |
          Execution_Block D_t D_v D_sp Statements
```

This prevented us from mixing things up by statically checking that we are constructing our rules in a properly typed manner. Furthermore, the implementation process of the programming language made us aware that certain things would be helpful when added to the specification such as adding the maps to the environment definition.

We present an example of Haskell implementation for the NS rules of Composition and Assign in Code 3. Recall the definition of these rules in Section 3.2.4.

Code 3 Haskell code for the composition rule and variable assignment.

```
evalStatements env (Composition st1 st2) = let
                                            env' = evalStatements env st1
                                            env'' = evalStatements env' st2
                                           in env''
evalStatements env (AssignVar x e) = let k = evalExpr env e in
                    if variable_type k env x then
                        upd_v env x k
                    else
                        StError Constraint_Error
```

Given Code 2 and 3 we can show how we define in Haskell the example of Max (in Code 1). We show this in Code 4.

Code 4 Haskell Code for the Max Program.

```
executeMax = Composition (AssignVar (ID_v "x") (Atom (TInt 9001)))
                (Composition (AssignVar (ID_v "y") (Atom (TInt 0)))
                (AssignVar (ID_v "res")
                (FCall (ID_f "Max")
                [(Var (ID_v "x") []),(Var (ID_v "y") [])]))))

bodyMax = IfThenElse (GTE (Var (ID_v "a") []) (Var (ID_v "b") []))
            (AssignVar (ID_v "aux") (Var (ID_v "a") []))
            (AssignVar (ID_v "aux") (Var (ID_v "b") []))

funMax = Df (ID_f "Max")
           ([ID_v "a", ID_v "b"], V funS)
           (ContractFunction Empty_Assert Empty_Assert)
           (ID_t "Short") (Dv [ID_v "aux"] (ID_t "Short") End_D_v) End_D_f
           bodyMax (Var (ID_v "aux") []) End_D_f

testMax = Execution_Block (Dt (ID_t "Short") (RInt ( 65536) 65536) End_D_t)
            (Dv [ID_v "x", ID_v "y", ID_v "res"] (ID_t "Short") End_D_v)
            (D_spf funMax) executeMax
```

As simple as the Max program may be, the constructions for it are quite big. This is of little importance as such code will rarely be seen by a human; this is an intermediate/embedded representation of our language. Due to it being highly structured and strongly typed, it is easier to generate the corresponding Haskell code from an mSPARK program, with the added benefit that the result of the parsing must yield a typable program.

As a side note, we decided to call the interpreter Hibana-chan since this is a very “young” interpreter for the first version of the mSPARK language, Hibana.

4 Program Logic of mSPARK

In this chapter we define the program logic for mSPARK as a Hoare-like logic.

The program logic is defined via a proof system and it handles the treatment of errors in expression evaluation. We show that the program logic is sound in relation to the operational semantics that we have defined. In our program logic, we only consider statements that are meant to be executed, that is, we assume that our environment has been previously enriched with all information related to types, variables and subprograms.

4.1 A Program Logic for mSPARK

Our program logic for mSPARK is presented by means of a proof system whose assertions are Hoare triples $\{\phi\}S\{\psi\}$. In Hoare triples, ϕ and ψ range over assertions, which in this chapter will be enriched with safe predicates, to explicitly treat possible errors in the evaluation of expressions. The component S refers to mSPARK statements, that in this chapter have only a restricted form.

One reason for leaving out some statements that we considered in the operational semantics is because, as we have seen then, we rewrite all executable statements in Inference Rules 3 into those simpler statements of Inference Rules 1 and 2.

The inference rules of our program logic are given in Inference Rules 11.

While most presentations of Hoare-like logics that can be found in the literature have a consequence rule, so that it is easier to do manual proofs, this rule also introduces difficulties, namely for the design of $VCGens$. We leave the consequence rule out of our system, but then the other inference rules have to compensate for it. Another departure point from traditional presentations of Hoare logic is the presence of safe predicates in side conditions of some inference rules. As mentioned before, this has to do with our interest in having our Hoare logic treating possible errors in the evaluation of expressions, an issue typically not addressed in traditional presentations of Hoare logic.

We use the notation $\vdash \{\phi\}S\{\psi\}$ with the meaning that the Hoare triple $\{\phi\}S\{\psi\}$ is derivable, i.e. has a derivation, using the inference rules of our program logic. As usual, a derivation of an Hoare triple $\{\phi\}S\{\psi\}$ is a tree of Hoare triples, built by using the inference rules Inference Rules 11, whose leaves must be axioms and whose root is $\{\phi\}S\{\psi\}$.

Before establishing the main result we prove about our proof system, the soundness theorem, we fix the precise set of expressions and assertions (as well as the syntax) we consider throughout this chapter. We do this in Definitions 1 and 2. The set of expressions is a subset of the expressions allowed in mSPARK; we only consider boolean, integer and array expressions in this chapter, though what we do in this chapter should extend to other mSPARK-expressions with no major difficulties. Some changes in syntax in this chapter, relative to the syntax of mSPARK, have to do with lightening of syntatic aspects or with having a syntax closer to new notation introduced in this chapter.

In this chapter we need to make precise what is the interpretation of an expression for a given environment. We do it in Definition 3. This notion is meant to agree with the notion $e \Downarrow_{\mathrm{Expr}(env)}$ used in the operational semantics (but notice we adopt here a different notation).

The set of assertions in this chapter enriches the one considered for the operational semantics with safe predicates. Given an environment *env* and an assertion ϕ, we use the notation $[\![\phi]\!](env)$ to denote the interpretation ($\top$ or $\bot$) of ϕ under *env*. The interpretation of the safe predicates for a given environment is presented in Definition 4. In this definition, compatible_types checks if two constants are compatible (integers with integers and booleans with booleans), safe_range checks if a given constant k is in the range of variable v (for arrays this means checking the range of the values in the array; this is equivalent to the *type_array_val* given in the operational semantics) and $\mathrm{safe}_{range\_ind}$ checks if a given constant k is in range for the representation type of index i of array a. Our treatment of safe predicates is inspired by the work of John C. Reynolds where it is called as "enabling predicate" [Rey04, Rey99] and by the work presented in the book [AFPMDS10]. The interpretation of assertions other than the safe predicates is assumed to be done in the expected way, as in first-order logic.

As compared to the notion of environment in Chapter 3, note that the relevant parts of the environment for interpreting expressions and assertions are the variables (and their state) and the information on representation types.

A key property required from the safe predicates is the following:

Property 1 (Safe property)**.**

$$\textit{If } [\![safe(e)]\!](env) = \top \textit{ then } [\![e]\!](env) \neq \textbf{\textit{error}}$$

Proof. The safe property can be proved by induction on the expression e. We present below some cases.

$\boxed{\text{Case } e = k}$

$[\![k]\!](env) = k \neq \textbf{error}$

$\boxed{\text{Case } e = -e'}$

$[\![e']\!](env) = \textbf{error}$ cannot happen as the hypothesis implies $[\![safe(e')]\!](env) = \top$ and then the induction hypothesis gives $[\![e']\!](env) \neq \textbf{error}$.

Therefore, $[\![-e']\!](env) = -[\![e']\!](env) \neq \textbf{error}$.

$\boxed{\text{Case } e = e_1/e_2}$

Hypotheses:
(i) $[\![safe(e_1)]\!](env) = \top$
(ii) $[\![safe(e_2)]\!](env) = \top$
(iii) $[\![e_2]\!](env) \neq 0$
(iv) compatible_types($[\![e_1]\!](env), [\![e_2]\!](env)$)

By the induction hypotheses, $[\![e_1]\!](env) \neq \textbf{error}$ and $[\![e_2]\!](env) \neq \textbf{error}$. From this, (iii) and (iv) it follows that $[\![e_1 \ / \ e_2]\!](env) \neq \textbf{error}$.

$\square$

Definition 1 Restricted expressions.

$\langle op \rangle$ ::= '+' | '-' | '*' | '<' | '<=' | '>' | '>=' | '=' | '/='

$\langle e \rangle$ ::= k | x | -$\langle e \rangle$ | $\langle e \rangle$ $\langle op \rangle$ $\langle e \rangle$ | $\langle e \rangle$ / $\langle e \rangle$ | $a(\langle e \rangle_1 \ldots \langle e \rangle_n)$, where k is a meta-variable ranging over integers and booleans.

$\langle be \rangle$ ::= NOT $\langle be \rangle$ | $\langle be \rangle$ AND $\langle be \rangle$ | $\langle be \rangle$ OR $\langle be \rangle$

Definition 2 Assertions.

$\langle \phi, \psi \rangle$ ::= $\langle be \rangle$ | safe($\langle e \rangle$) | safe_range($\langle v \rangle$,$\langle e \rangle$) | $\neg\phi$ | $\phi \,\square\, \psi$ | $\forall x \,.\, \phi$ | $\exists x \,.\, \phi$
where $\square = \{\wedge, \vee, \rightarrow, \leftrightarrow\}$

As we have seen in the operational semantics, we rewrite all executable statements in Inference Rules 3 into those simpler statements of Inference Rules 1 and 2. As such, we shortened our program logic to account only for these cases, in Inference Rules 11.

While most Hoare-like logics that can be found in the literature have a consequence rule,

Definition 3 Expression Evaluation for Restricted Expressions.

$$[\![k]\!](env) = k$$
$$[\![-e]\!](env) = -[\![e]\!](env)$$

$$[\![e_1 \text{ op } e_2]\!](env) = [\![e_1]\!](env) \text{ op } [\![e_2]\!](env), \text{ if}$$
$$\text{compatible\_types}([\![e_1]\!](env), [\![e_2]\!](env))$$

$$[\![e_1 \ / \ e_2]\!](env) = [\![e_1]\!](env) \ / \ [\![e_2]\!](env), \text{ if}$$
$$\text{compatible\_types}([\![e_1]\!](env), [\![e_2]\!](env)) \text{ and } [\![e_2]\!](env) \neq 0$$

$$[\![\text{not } be]\!](env) = \neg[\![be]\!](env)$$
$$[\![be_1 \text{ and } be_2]\!](env) = [\![be_1]\!](env) \text{ and } [\![be_2]\!](env)$$
$$[\![be_1 \text{ or } be_2]\!](env) = [\![be_1]\!](env) \text{ or } [\![be_2]\!](env)$$

Definition 4 Safe predicate interpretation.

$$\begin{aligned}
[\![\text{safe}(k)]\!](env) &= \top \\
[\![\text{safe}(x)]\!](env) &= \top \\
[\![\text{safe}(-e)]\!](env) &\iff [\![\text{safe}(e)]\!](env) = \top \\
[\![\text{safe}(e_1 \ \Box \ e_2)]\!](env) &\iff [\![\text{safe}(e_1)]\!](env) \wedge [\![\text{safe}(e_2)]\!](env) \wedge \\
&\quad\ \text{compatible\_types}([\![e_1]\!](env), [\![e_2]\!](env)) \\
&\quad\ \text{where } \Box \in \{+, -, *, =, <, <=, >, >=, /=\} \\
[\![\text{safe}(e_1 \ / \ e_2)]\!](env) &\iff [\![\text{safe}(e_1)]\!](env) \wedge [\![\text{safe}(e_2)]\!](env) \wedge \\
&\quad\ \text{compatible\_types}([\![e_1]\!](env), [\![e_2]\!](env)) \\
&\quad\ \wedge [\![e_2]\!](env) \neq 0, \text{ where } [\![e_2]\!](env) :: \textit{integer} \\
[\![\text{safe}(\textbf{not} \ \ b)]\!](env) &\iff [\![\text{safe}(b)]\!] \\
[\![\text{safe}(b_1 \textbf{ and } b_2)]\!](env) &\iff [\![\text{safe}(b_1)]\!](env) \wedge [\![\text{safe}(b_2)]\!](env) \\
[\![\text{safe}(b_1 \textbf{ or } b_2)]\!](env) &\iff [\![\text{safe}(b_1)]\!](env) \wedge [\![\text{safe}(b_2)]\!](env) \\
[\![\text{safe}(a(e_1, \ldots, e_n))]\!](env) &\iff \bigwedge_{i=1}^{n} ([\![\text{safe}(e_i)]\!](env) \wedge [\![\text{safe}_{range\_ind}(a, i, e_i)]\!](env)) \\
[\![\text{safe\_range}(v, e)]\!](env) &\iff [\![e]\!](env) :: \textit{variable\_type}(v, env)
\end{aligned}$$

so that it is easier to do manual proofs, this rule also introduces ambiguity in our deductive system.

For our program logic, we removed the consequence rule and have rewritten the logic in a goal-oriented fashion so that it is easier to generate VCs automatically from it. We present a proof of soundness for these rules in the next chapter.

Inference Rules 11 Program Logic for Basic Statements of mSPARK.

$$\frac{}{\{\phi\}\ \mathbf{null}\ \{\psi\}} \qquad \text{(NULL)}$$

if $\models \phi \to \psi$

$$\frac{}{\{\phi\}\ x := e\ \{\psi\}} \qquad \text{(ASSIGN)}$$

if $\models \phi \to (\text{safe}(e) \wedge \text{safe\_range}(x, e))$ and $\models \phi \to \psi[e/x]$

$$\frac{}{\{\phi\}\ a(e_1, \ldots, e_n) := e\ \{\psi\}} \qquad \text{(ASSIGN ARRAY)}$$

if $\models \phi \to (\text{safe}(a(e_1, \ldots, e_n)) \wedge \text{safe}(e) \wedge \text{safe\_range}(a, e))$ and $\models \phi \to \psi[a(e_1, \ldots, e_n \rhd e)/a]$

$$\frac{\{\phi\}\ S_1\ \{\psi\} \qquad \{\phi\}\ S_2\ \{\psi\}}{\{\phi\}\ S_1\ ;\ S_2\ \{\psi\}} \qquad \text{(SEQUENCE)}$$

$$\frac{\{\phi \wedge b\}\ S_t\ \{\psi\} \qquad \{\phi \wedge \neg b\}\ S_f\ \{\psi\}}{\{\phi\}\ \mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f\ \mathbf{end\ if}\ \{\psi\}} \qquad \text{(IF)}$$

if $\models \phi \to \text{safe}(b)$

$$\frac{\{\theta \wedge b\}\ S\ \{\theta\}}{\{\phi\}\ \mathbf{while}\ b\ \theta\ \mathbf{loop}\ S\ \mathbf{end\ loop}\ \{\psi\}} \qquad \text{(WHILE)}$$

if $\models \phi \to \theta$ and $\models \theta \to \text{safe}(b)$ and $\models \theta \wedge \neg b \to \psi$

In Inference Rules 12 we present the program logic for subprogram invocation where the suprograms have been annotated with contracts. We use the following notions:

- π establishes the correction for the set of subprograms in our program. That is, it takes all subprograms in a given program and establishes the correction for it, based on the body of the subprogram and its pre- and post-conditions;
- $\mathbf{pre}(sp)$ fetches the pre-condition of subprogram sp from the environment;
- $\mathbf{post}(sp)$ fetches the post-condition of subprogram sp from the environment;
- frame(p) fetches the list of global variables (frame condition) of procedure p from the environment;
- result(f) fetches the result variable of function f from the environment;

Also note that we only handle function calls when they are assigned to a variable. This may seem limited but given the order of expression evaluations, we can rewrite the expression into a series of assignments and then "glue" back together the expression into a single assignment. This is one way to overcome the problem of only having function calls with assignment.

While there are tools, such as ACSL and SPARK, which address the problem of having a program logic for subprograms with contracts, we were not able to find any work on proving the soundness of those rules.

There exists related work, dealing with Hoare-like logics and the adaptation problem, such as the work by Kleymann [Kle99], but it is outside the scope of contracts.

That being the case, we were unable to achieve results at the time of the writing of the thesis related to the soundness of these rules, so we will not present proofs related to their soundness. We have mainly followed the work that has been done in [AFPMDS10].

Inference Rules 12 Program Logic for Subprogram Invocation of mSPARK.

$$\frac{\{\pi\}}{\{\phi\}\ p\ \{\psi\}}\ (\textsc{Parameterless Procedure Invocation})$$

if $\models \phi \rightarrow \forall x_f.(\mathbf{pre}(p) \rightarrow \lfloor \mathbf{post}(p)[x_f/\text{frame}(p)] \rfloor) \rightarrow \psi[x_f/\text{frame}(p)]$ and $\models \phi \rightarrow \text{safe}(p)$

$$\frac{\{\pi\}}{\{\phi\}\ y := f\ \{\psi\}}\ (\textsc{Parameterless Function Invocation})$$

if $\models \phi \rightarrow (\mathbf{pre}(f) \rightarrow \mathbf{post}(f)) \rightarrow \psi[\text{result}(f)/y]$ and $\models \phi \rightarrow \text{safe}(f)$

$$\frac{\{\pi\}}{\{\phi\}\ y := f(e_1 \ldots e_n)\ \{\psi\}}\qquad (\textsc{Function Invocation})$$

if $\models \phi \rightarrow (\mathbf{pre}(f)[e_1 \ldots e_n/par_1 \ldots par_n] \rightarrow \mathbf{post}(f)[e_1 \ldots e_n/par_1 \ldots par_n]) \rightarrow$
$\psi[\text{result}(f)/y]$ and $\models \phi \rightarrow \text{safe}(f(e_1 \ldots e_n))$

4.2 Soundness

Soundness is the property that what can be stated in the axiomatic semantics agrees with the operational semantics.

Definition 5 Validity of Hoare triples.

We say a Hoare tripe $\{\phi\}\ S\ \{\psi\}$ is valid iff

$$\forall env \neq \mathbf{error}\ \forall env'.\ [\![\phi]\!](env) = \top \wedge (S, env) \rightsquigarrow env' \Rightarrow env' \neq \mathbf{error} \wedge [\![\psi]\!](env') = \top$$

We use the notation $\models \{\phi\}\ S\ \{\psi\}$ to mean that $\{\phi\}\ S\ \{\psi\}$ is valid.

Theorem 1 (Soundness of mSPARK's program logic)**.**

$$\vdash \{\phi\}\ S\ \{\psi\} \Rightarrow\ \models \{\phi\}\ S\ \{\psi\}$$

Proof. The proof of the result is by induction on the derivation of $\{\phi\}\ S\ \{\psi\}$.

Case 1 Null axiom.

$$\frac{}{\{\phi\}\ \mathbf{null}\ \{\psi\}} \qquad (\textsc{Null}) \quad \text{if} \models \phi \rightarrow \psi$$

We need to prove $\models \{\phi\}\ \mathbf{null}\ \{\psi\}$, i.e.

$\forall env \neq \mathbf{error}\ \forall env'\ .\ [\![\phi]\!](env) = \top \wedge (\mathbf{null}, env) \rightsquigarrow env' \Rightarrow env' \neq \mathbf{error} \wedge [\![\psi]\!](env') = \top$

Let $env \neq \mathbf{error}$ and env' be environments such that:
(i) $[\![\phi]\!](env) = \top$
(ii) $(\mathbf{null}, env) \rightsquigarrow env'$

So, $env = env'$ by the operational semantics (note that, as $env \neq \mathbf{error}$, (ii) only matches the rule (Null). Then it is immediate that $env' \neq \mathbf{error}$, as we assume $env \neq \mathbf{error}$.

From the side condition $\models \phi \rightarrow \psi$ and the facts (i) and $env = env'$, follows
$[\![\psi]\!](env') = [\![\psi]\!](env) = \top$.

Case 2 Assign axiom.

$$\frac{}{\{\phi\}\ x := e\ \{\psi\}} \qquad (\textsc{Assign})$$

$$\text{if} \models \phi \rightarrow (\text{safe}(e) \wedge \text{safe\_range}(x, e)) \text{ and } \models \phi \rightarrow \psi[e/x]$$

We want to prove $\models \{\phi\}\ x := e\ \{\psi\}$, i.e.

$\forall env \neq \textbf{error}\ \forall env'\ .\ [\![\phi]\!](env) = \top \wedge (x := e, env) \rightsquigarrow env' \Rightarrow env' \neq \textbf{error} \wedge [\![\psi]\!](env') = \top$

Let $env \neq \textbf{error}$ and env' be environments such that:
(i) $[\![\phi]\!](env) = \top$
(ii) $(x := e, env) \rightsquigarrow env'$

Remember: $[\![\text{safe\_range}(x, e)]\!](env) = \top \iff [\![e]\!](env) :: \mathit{variable\_type}(env, x)$.
From (1) and the side condition $\models \phi \rightarrow (\text{safe}(e) \wedge \text{safe\_range}(x, e))$ we know that
(iii) $[\![\text{safe}(e)]\!](env) = \top$ and
(iv) $[\![\text{safe\_range}(x, e)]\!](env) = \top$

Rule $AssignVar_{type\_error}$ of the operational semantics could not have been applied because (iv) guarantees $[\![e]\!](env) :: \mathit{variable\_type}(env, x)$.

So, $env' = env[[\![e]\!](env)/x]$.

From (iii) and (iv) it follows immediately that $env' \neq \textbf{error}$, since env is assumed to be well formed and the substitution $[e/x]$ is well defined.

From (i) and the side condition $\models \phi \rightarrow \psi[e/x]$, follows $[\![\psi[e/x]]\!](env) = \top$, but, by a typical substitution lemma (that one can shown by induction on ψ), this is equivalent to $[\![\psi]\!](env[[\![e]\!](env)/x]) = \top$, i.e. $[\![\psi]\!](env') = \top$.

Case 3 Assign Array Axiom.

$$\frac{}{\{\phi\}\ a(e_1,\ldots,e_n) := e\ \{\psi\}} \qquad (\textsc{Assign Array})$$
if $\models \phi \rightarrow (\text{safe}(a(e_1,\ldots,e_n)) \wedge \text{safe}(e) \wedge \text{safe\_range}(a,e))$ and $\models \phi \rightarrow \psi[a(e_1,\ldots,e_n \triangleright e)/a]$

We want to prove $\models \{\phi\}\ a(e_1,\ldots,e_n) := e\ \{\psi\}$, i.e.

Expanding:
$\forall env \neq \textbf{error}\ \forall env'\ .\ [\![\phi]\!](env) = \top \wedge (a(e_1,\ldots,e_n) := e, env) \rightsquigarrow env' \Rightarrow env' \neq \textbf{error} \wedge [\![\psi]\!](env') = \top$

Let $env \neq \textbf{error}$ and env' be environments such that:
(i) $[\![\phi]\!](env) = \top$
(ii) $(a(e_1,\ldots,e_n) := e,\ env) \rightsquigarrow env'$

From (1) and the side condition $\models \phi \rightarrow (\text{safe}(a(e_1,\ldots,e_n)) \wedge \text{safe}(e) \wedge \text{safe\_range}(a,e))$ we know that
(iii) $[\![\text{safe}(a(e_1,\ldots,e_n))]\!](env) = \top$
(iv) $[\![\text{safe}(e)]\!] = \top$
(v) $[\![\text{safe\_range}(a,e)]\!] = \top$

Rule AssignArr$_{index\_error}$ could have not been applied because (iii) guarantees $[k_1,\ldots,k_n] :: type\_array\_ind(env,a)$.

Rule AssignArr$_{value\_error}$ could have not been applied because (iv) guarantees $[\![\text{safe}(e)]\!] = \top$ and (v) guarantees $k :: type\_array\_val(env,a)$.

So $env' = env[a([k_1\ldots k_n] \triangleright k)/a]$. From (iii), (iv) and (v) it follows immediately that $env' \neq \textbf{error}$, since env is assumed to be well formed and the substitution $[a([k_1\ldots k_n] \triangleright k)/a]$ is well defined.

From (i) and the side condition $\models \phi \rightarrow \psi[a(e_1,\ldots,e_n \triangleright e)/a]$, follows $[\![\psi[a([k_1\ldots k_n] \triangleright k)/a]]\!](env) = \top$, but, by a typical substitution lemma (that one can shown by induction on ψ), this is equivalent to $[\![\psi]\!](env[[\![a([k_1\ldots k_n] \triangleright k)]\!](env)/x]) = \top$, i.e. $[\![\psi]\!](env') = \top$.

Case 4 Sequence rule.

$$\frac{\{\phi\}\ C_1\ \{\theta\} \qquad \{\theta\}\ C_2\ \{\psi\}}{\{\phi\}\ C_1\ ;\ C_2\ \{\psi\}} \qquad (\textsc{Sequence})$$

We want to prove $\models \{\phi\}\ C_1\ ;\ C_2\ \{\psi\}$, i.e.

$\forall env \neq \mathbf{error}\ \forall env'\ .\ [\![\phi]\!](env) = \top \wedge (C_1\ ;\ C_2, env) \rightsquigarrow env' \Rightarrow env' \neq \mathbf{error} \wedge [\![\psi]\!](env') = \top$

Let $env \neq$ **error** and env' be environments such that:
(i) $[\![\phi]\!](env) = \top$
(ii) $(C_1\ ;\ C_2, env) \rightsquigarrow env'$

As $env \neq$ **error**, in the operational semantics only rule (Composition) could have been used to derive (ii). So, there exists env'' such that:
(iii) $(C_1, env) \rightsquigarrow env''$
(iv) $(C_2, env'') \rightsquigarrow env'$

Using the induction hypothesis relative to the first premise, combined with (i) and (iii), one gets $env'' \neq$ **error** and $[\![\theta]\!](env'') = \top$. In turn, these two facts and (iv), combined with the induction hypothesis relative to the second premise, yield, as wanted, $env' \neq$ **error** and $[\![\psi]\!](env') = \top$.

Case 5 If rule.

$$\frac{\{\phi \wedge b\}\ S_t\ \{\psi\} \qquad \{\phi \wedge \neg b\}\ S_f\ \{\psi\}}{\{\phi\}\ \mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f\ \mathbf{end\ if}\ \ \{\psi\}} \quad (\textsc{If})$$

if $\models \phi \rightarrow \text{safe}(b)$

We want to prove $\models \{\phi\}$ **if** b **then** C_t **else** C_f **end if** $\{\psi\}$, i.e.

$\forall env \neq \mathbf{error}\ \forall env'\ .\ [\![\phi]\!](env) = \top \wedge (\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\ \mathbf{end\ if}\ , env) \rightsquigarrow env' \Rightarrow env' \neq \mathbf{error} \wedge [\![\psi]\!](env') = \top$

Let $env \neq$ **error** and env' be environments such that:
(i) $[\![\phi]\!](env) = \top$
(ii) $(\{\phi\}\ \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\ \mathbf{end\ if}\ \ \{\psi\}, env) \rightsquigarrow env'$

The proof proceeds by case analysis on $[\![b]\!](env)$.

$\boxed{\text{Case } [\![b]\!](env) = \top.}$

By the operational semantics, it must be the case that.
(iii) $(C_t, env) \rightsquigarrow env'$

Combining (i) with our case assumption we have $[\![\phi \wedge b]\!](env) = \top$. This, together with (ii) and the induction hypothesis relative to the first premise gives, as wanted, $env' \neq$ **error** and $[\![\psi]\!](env') = \top$.

$\boxed{\text{Case } [\![b]\!](env) = \bot.}$

This case follows similarly. By the operational semantics, we must have
(iii) $(C_f, env) \rightsquigarrow env'$

Combining (i) with our case assumption we have $[\![\phi \wedge \neg b]\!](env) = \top$, which together with (ii) and the induction hypothesis relative to the second premise gives $env' \neq$ **error** and $[\![\psi]\!](env') = \top$.

$\boxed{\text{Case } [\![b]\!](env) = \mathbf{error}.}$

Combining (i) with the side condition, follows $[\![\text{safe}(b)]\!](env) = \top$ and this, by the safe property 1 implies that $[\![b]\!](env) \neq$ **error**. So, the case $[\![b]\!](env) =$ **error** cannot actually occur.

Case 6 While rule.

$$\frac{\{\theta \wedge b\}\ S\ \{\theta\}}{\{\phi\}\ \textbf{while}\ b\ \theta\ \textbf{loop}\ S\ \textbf{end loop}\ \{\psi\}} \qquad (\textsc{While})$$

if $\models \phi \rightarrow \theta$ and $\models \theta \rightarrow \text{safe}(b)$ and $\models \theta \wedge \neg b \rightarrow \psi$

We want to prove

$\forall env \neq \textbf{error}\ \forall env'\ .\ [\![\phi]\!](env) = \top \wedge (\textbf{while}\ b\ \theta\ \textbf{loop}\ S\ \textbf{end loop}, env) \rightsquigarrow env' \Rightarrow env' \neq \textbf{error} \wedge [\![\psi]\!](env') = \top$

Let us have $env \neq \textbf{error}$ and env' environments such that:
(i) $[\![\phi]\!](env) = \top$
(ii) $(\textbf{while}\ b\ \theta\ \textbf{loop}\ S\ \textbf{end loop},\ env) \rightsquigarrow env'$

From (i) and the side condition $\models \phi \rightarrow \theta$ follows (iii) $[\![\theta]\!](env) = \top$
Claim: $\forall env'' \neq \textbf{error}([\![\theta]\!](env'') = \top \wedge\ (\textbf{while}\ b\ \theta\ \textbf{loop}\ S\ \textbf{end loop},\ env'') \rightsquigarrow env' \ \Rightarrow env' \neq \textbf{error} \wedge [\![\theta \wedge \neg b]\!](env') = \top)$

Note that from the claim, $env \neq \textbf{error}$, (iii) and (ii), follows $env' \neq \textbf{error}$ and $[\![\theta \wedge \neg b]\!](env') = \top$, and the latter together with the side condition $\models \phi \rightarrow \theta$ gives $[\![\psi]\!](env') = \top$.
Now we prove the claim; it is done by induction on the operational semantics relation $\rightsquigarrow$.

$env'' \neq \textbf{error}$ and assume
(i) $[\![\theta]\!](env'') = \top$
(ii) $(\textbf{while}\ b\ \theta\ \textbf{loop}\ S\ \textbf{end loop},\ env'') \rightsquigarrow env'$

The proof proceeds by an analysis of $[\![b]\!](env'')$.

$\boxed{\text{Case } [\![b]\!](env'') = \bot}$

The operational semantics tells that env'' must be env'. Thus immediately $env' \neq \textbf{error}$ as $env'' \neq \textbf{error}$ and combining (i) with the case assumption gives $[\![\theta \wedge \neg b]\!](env'') = \top$.

$\boxed{\text{Case } [\![b]\!](env'') = \top}$

From the operational semantics we know that:
(iii) $(C,\ env'') \rightsquigarrow env_0$ for some env_0
(iv) $(\textbf{while}\ b\ \theta\ \textbf{loop}\ S\ \textbf{end loop},\ env_0) \rightsquigarrow env'$

The outer induction hypothesis gives us $\models \{\theta \wedge b\}C\{\theta\}$, i.e.
$\forall env_1 \neq \textbf{error}\ \forall env_2\ .\ [\![\phi]\!](env) = \top \wedge (C_1, env_1) \rightsquigarrow env_2 \Rightarrow env_2' \neq \textbf{error} \wedge [\![\psi]\!](env_2) = \top$

So, for $env'' \neq \textbf{error}$, (i), the case assumption and (iii), we get $env_0 \neq \textbf{error}$ and $[\![\theta]\!](env_0) = \top$. Now we simply need to use the inner induction hypothesis, together with these two observations and (iv), to conclude $env' \neq \textbf{error}$ and $[\![\theta \wedge \neg b]\!](env') = \top$.

$\boxed{\text{Case } [\![b]\!](env'') = \textbf{error}}$

From (i) and the side condition $\models \theta \rightarrow \text{safe}(b)$ follows $[\![\text{safe}(b)]\!](env'') = \top$. But then, the safe property 4.1 gives that $[\![b]\!](env'') \neq \textbf{error}$. So, the case $[\![b]\!](env'') = \textbf{error}$ cannot occur.
The proof of the claim is thus finished.

This case of the While rule completes the proof of soundness. ■

4.3 Verification Condition Generator for mSPARK Hibana

The purpose of the VCGen algorithm is to generate a set of formulas where the validity of those formulas implies the validity of a Hoare triple. This is a general property, common to all VCGen algorithms.

This is done by calculating the side conditions of a derivation, where the Hoare triple from the program logic in Section 4 is in the conclusion. The algorithm computes those conditions for a tree that is not built explicitly.

The algorithm uses the notion of weakest precondition [Dij75] to instantiate the antecedents of the side conditions (which are all implications) of the rule's application when the rule's antecedent is not in the conclusion of the rule (note that in the While rule the antecedent of two of the side conditions is fixed).

The definition of the VCGen could also be written in a recursive style directly in the structure of the statements, keeping the existing pre-condition, but instead we chose to use an auxiliary recursive function VC, that does not take the pre-condition as a parameter, which allows to exclude from the set of generated VCs several tautologies that would then be introduced; this is in fact an optimization.

Definition 6 contains the definition of a VCGen for mSPARK; given a Hoare triple $\{\phi\}C\{\psi\}$, VCG($\{\phi\}C\{\psi\}$) is indeed a set of first-order formulas whose validity implies the validity of the triple. Although we leave the proof of soundness of the algorithm for future work, we note that since we have removed the consequence rule from the program logic (and thus removed the ambiguity in the choice of the rules to be applied) and written the inference system in a goal-oriented style, the correspondence between the program logic and the VCGen algorithm is almost immediate.

This algorithm should be applied not only to the **execute** part of our program but also to all subprogram declarations so that we can obtain π (recall the definition of π from the previous section).

Definition 6 Weakest Preconditions, Verification Conditions and VCGen.

$\mathbf{wp}(\mathbf{null}, \psi) = \psi$

$\mathbf{wp}(x := e, \psi) \;=\; \text{safe}(e) \wedge \psi[e/x]$

$\mathbf{wp}(a(e_1, \ldots, e_n) := e, \psi) \;=\; \text{safe}(a(e_1, \ldots, e_n \rhd e)) \wedge \psi[e/a]$

$\mathbf{wp}(C_1; C_2, \psi) \;=\; \mathbf{wp}(C_1, \mathbf{wp}(C_2, \psi))$

$\mathbf{wp}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, \psi) = \text{safe}(b) \wedge (b \to \mathbf{wp}(C_t, \psi)) \wedge (\neg b \to \mathbf{wp}(C_f, \psi))$

$\mathbf{wp}(\mathbf{while}\ b\ \theta\ \mathbf{loop}\ C\ \mathbf{end\ loop}, \psi) = \theta$

$\mathbf{wp}(p, \psi) = \forall x_f.(\mathbf{pre}(p) \to \lfloor \mathbf{post}(p)[x_f/frame(p)] \rfloor) \to \psi[x_f/frame(p)]$

$\mathbf{wp}(y := f, \psi) = (\mathbf{pre}(f) \to \mathbf{post}(f)) \to \psi[result(f)/y]$

$\mathbf{wp}(y := f(e_1 \ldots e_n), \psi) = (\mathbf{pre}(f)[e_1 \ldots e_n/par_1 \ldots par_n] \to$
$\mathbf{post}(f)[e_1 \ldots e_n/par_1 \ldots par_n]) \to \psi[\text{result}(f)/y]$

$\mathbf{VC}(\mathbf{null}, \psi) = \emptyset$

$\mathbf{VC}(x := e, \psi) = \emptyset$

$\mathbf{VC}(x(e_1, \ldots, e_n) := e, \psi) = \emptyset$

$\mathbf{VC}(C_1; C_2, \psi) = \mathbf{VC}(C_1, \mathbf{wp}(C_2, \psi)) \;\cup\; \mathbf{VC}(C_2, \psi)$

$\mathbf{VC}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\ \mathbf{end\ if}\ , \psi) = \mathbf{VC}(C_t, \psi) \cup \mathbf{VC}(C_f, \psi)$

$\mathbf{VC}(\mathbf{while}\ \theta\ b\ \mathbf{loop}\ C\ \mathbf{end\ loop}, \psi) =$
$\{\theta \to safe(b), (\theta \wedge b) \to \mathbf{wp}(C, \theta), (\theta \wedge \neg b) \to \psi\} \cup \mathbf{VC}(C, \theta)$

$\mathbf{VC}(p, \psi) = \emptyset$

$\mathbf{VC}(y := f, \psi) = \emptyset$

$\mathbf{VC}(y := f(e_1 \ldots e_n), \psi) = \emptyset$

$\mathbf{VCG}(\{\phi\}\ C\ \{\psi\}) = \{\theta \to \mathbf{wp}(C, \psi)\} \cup \mathbf{VC}(C, \psi)$

---

5 Conclusion and Future Work

We now present the results and achievements we were able to obtain during the research and development of the thesis.

5.1 Conclusion

We believe that the thesis has been very productive, at the educational level and at the scientific level. We have experienced the difficulties of formalizing a real programming language in a rigorous manner and have been able to use those lessons to further the development of the programming language, while identifying future work that we wish to build upon what we already have. Our most significant contributions are:

- The publication of a comparative case study in a top tier conference;
- The creation of a program logic for a programming language with safety restrictions and range types;
- A verification condition generation algorithm for the language;
- Operational Semantics for this subset of SPARK;
- Proved the soundness of our program logic regarding a relevant subset of our programming language.

Furthermore, besides the theoretical aspects, we have created an interpreter for our programming language by following the formal semantics that we had defined. This implementation also allowed us to give feedback on details that would benefit the specification, especially when trying to do an implementation of the programming language. It is our wish not only to develop theories and methodologies for program verification but also tools and methods that can be used in practice.

We have also been successful in publishing articles with the results of our research. We intend to continue developing this project with further work on formal methods for safety-critical systems since this is a very active area, with several interesting challenges that remain to be solved and where we believe we are able to make interesting contributions.

5.2 Future Work

In this section we present work we intend to develop and we suggest some possible future directions, based on the proposed mSPARK platform.

Extending the mSPARK Language SPARK has proven itself to be very useful in the safety-critical domain by providing a restricted subset of Ada that is easier to verify. This success in SPARK verification is in part due to the removal of some hard verification problems, such as heap manipulation, and by having the source code annotated with helpful indications regarding information-flow and frame conditions.

By extending the language, this easier verification process may be lost. One way to avoid this is to provide an equivalent but alternative language, such as Hibana, which is openly directed towards research problems and the formal verification area. If the solutions are deemed useful for the safety-critical arena, they can eventually be adopted by practitioners in industry.

Extending the language has two meanings. One is to extend the programming language; the other is to extend the BISL. While extending the programming language may eventually lead to extending the BISL, we see fit to extend certain aspects of the existing BISL, even without making changes to the programming language. Future additions to the BISL may include concurrency constructs, timing constructs, as well as adding new ways to define mathematical properties (we look forward to being able to write inductive definitions in Hibana; this is an absent mathematical mechanism in SPARK and Hibana).

Furthermore, while SPARK has support for concurrent and real-time constructs (using the RavenSPARK profile), Hibana has none. As a first step, we intend to focus on concurrency and, following our methodology, it seems obvious that we need to adapt our operational semantics to include features from RavenSPARK. There is related work on the operational semantics of Ravenscar [HN08] which we believe we can adapt onto our semantics.

RavenSPARK is a set of restrictions on top of Ravenscar. Ravenscar [Bur99] is a set of programming constructs for developing real-time concurrent software for safety-critical systems. It is a subset of Ada's tasking facilities, with several features removed and/or limited, that is more amenable for analysis and predictability. Given its features, it seems to provide a good starting point for research on program verification with concurrency, as SPARK is a good starting point for program verification for sequential constructs.

Although the operational semantics of concurrent constructs is well-known in the literature, its axiomatic treatment is rather lacking. We believe it will be possible to use Hibana and its operational semantics with concurrency to further develop the axiomatic treatment. This will bring many exciting challenges and will lead to research on newer logics such as Concurrent Separation Logic [FFS07].

Implementing More Tools for mSPARK We have presented in Chapter 3, along with the methodology, a set of tools that could be implemented, such as the SPARK adapter, the mSPARK parser and a multi-prover VCGen. While this thesis provides several theoretical bases to enable such work, the tools are yet incomplete/unimplemented. One of the first points we would like to continue our work on, is the implementation of such tools.

As we stated before, we are interested in having practical tools as well as theoretical bases for them. With this in mind, we should develop further work on the implementation of such tools in the near future.

Completing the Proofs Proofs have been presented for several parts of our work. Although we show the soundness of our program logic and show examples of proofs, for instance, for the safe property of expressions, this is done to subsets of our language that were manageable to do in the available time. Further work will need to be done to complete this proofs, especially related to the VCGen that, although its relation with the program logic is almost immediate, it still has to be shown.

5.3 Possible Applications for the mSPARK framework

In the previous section we have presented work that we wanted to do (and that we intend to complete) during the development of the thesis. This section focuses on some possibilities and ideas that were "discovered" while doing research for the thesis.

Designing a Formal Static Semantics for mSPARK We have presented a informal static semantics in this thesis that was enough to provide some intuition on the assumptions that were being made while developing the natural semantics and the program logic. The next logical step would be to formally specify the static semantics of mSPARK and to develop an Examiner-like tool (XStatic?) to deal with mSPARK source code.

This will eventually be needed, especially when starting to deviate from regular SPARK by introducing new features to the language. In the meanwhile, using the Examiner and a source code adapter is sufficient.

Implementing the VACID-0 Benchmark in mSPARK VACID-0 [LM10] was proposed as a set of challenges for formal verification of algorithms and data structures. This acts as a benchmark for program verification technologies and their ability to deal with the implementation details required from those data structures and the properties we want to check on the data structures.

As work on mSPARK evolves, especially on the toolset, we see this as a possibility of implementing these challenges as a way of showing the validity of our work and its applicability.

Automatic Generation of Test Suites Another possibility we see for mSPARK is that, since the language is relatively small, and since it has all this knowledge of types (atom types, representation types) and assertions, it would be ideal to automatically generate test suites for it. Pre- and post-conditions could also be used to further restrict the testing (pre-conditions) and validate outputs (post-conditions). It should be mentioned that the Hi-Lite project is also working in this area.

Mechanical Proofs The proofs that we have shown in the thesis were all done manually. To further validate our work and to have a more thorough and verifiable approach to the proof process, mechanical proofs should be provided, with the aid of a proof assistant, such as Coq [dt04]. This process is very demanding since we have to introduce all theoretical work that was developed into the proof assistant (which is in itself a validation of our work) and then to repeat the proofs that we already managed to do but in a more rigorous manner.

Proof Carrying Code Generation As it has already been stated, safety is a major concern for SPARK. Although, by design, SPARK enables us to construct safe software, when we reach the binary level, much information is left out. Proof Carrying Code (PCC) is a way to ensure that our binaries have certain safety properties and that it is safe to execute them.

One of the difficulties with having a PCC platform is to have a formal semantics of the source code language, which is usually unavailable, and of the language used by the untrusted binary code (e.g. machine code of x86, Motorola 68k, etc). With mSPARK we aim at providing an open formal semantics for the source code language. Also, recently GNAT GPL 2010 has added support for the AVR 8-bit microprocessor. This microprocessor is used in some safety-critical arenas such as the nuclear and automotive industries. The microprocessor is a modified Harvard architecture with a RISC ISA (Instruction Set Architecture). We believe that this can be used as a case study for a PCC platform, with potential industrial application, because of the microprocessor itself and because its reduced ISA facilitates the development of its formal semantics.

From Alloy to Z and mSPARK Alloy is a lightweight formal method that is used to model software, protocols, systems and so on. Alloy uses a relational logic which makes it lightweight to think in terms of connections between objects and restrictions upon their use. The Z notation is one of the oldest and most well-known formal methods. Altran Praxis is one of the users of Z notation in their software engineering process, which they call REVEAL.

As stated in [MGL10] it is possible to map Z specifications to Alloy models, although some things, like schemas, are difficult to reason about. We propose the inverse; since Alloy allows us to rapidly think about our meta-model and find counter-examples to it, displaying its counter-examples in a visual representation, it seems rather appropriate to translate Alloy into Z as well. Furthermore, given the capabilities of the Alloy language, we could potentially generate source code stubs in SPARK/Hibana with the meta-model information. In one step, we would provide Z notation specifications as well as mSPARK/SPARK source code stubs with, at least, contracts already specified.

Furthermore, Alloy counter-examples may be used to generate unit tests for our implementations.

Floating-point Computations and Gappa Traditionally, the area of safety-critical software has used fixed-point computations instead of floating-point. This is because it is easier to calculate and prove the upper bound error and the propagation of errors in calculations, without having to account for problems with the internal representations of floating-points, among other problems.

Meanwhile, practitioners are starting to feel the need to use floating-point computations, despite the difficulties [Mon08] that its use brings along.

Gappa [BFM09c] is a new attempt at providing automatic proof support for programs that use floating-point computations. We believe that the incorporation of such a tool into mSPARK would provide great added value to our language and toolset.

Bibliography

[Abr96] J.-R. Abrial. The B-book: assigning programs to meanings. Cambridge University Press, New York, NY, USA, 1996.

[AD06] Peter Amey and Bernard Dion. Combining model-driven design with diverse formal verification. January 2006.

[AFPMDS10] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo De Sousa. Rigorous Software Development: An Introduction to Program Verification. Undergraduate Topics in Computer Science. Springer, 1^{st} edition, December 2010.

[Alt] Altran Praxis. New lunar lander project relies on SPARK programming language.

[AR02] Lilian Burdy Antoine and Antoine Requet. JACK: Java Applet Correctness Kit. In In Proceedings, 4th Gemplus Developer Conference, 2002.

[BA93] M. Ben-Ari. Mathematical logic for computer science. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[Bar08] John Barnes. Safe and Secure Software: An invitation to Ada 2005. AdaCore, 2008.

[BCC$^+$03] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan, M. Leino, and Erik Poll. An overview of JML tools and applications, 2003.

[BCJ$^+$06] J Barnes, R Chapman, R Johnson, J Widmaier, D Cooper, and B Everett. Engineering the Tokeneer enclave protection software. In IEEE International Symposium on Secure Software Engineering (ISSSE). IEEE Press, 2006.

[BFM$^+$09a] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C Specification Language, version 1.4, 2009. http://frama-c.cea.fr/acsl.html.

[BFM$^+$09b] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto.

ACSL: ANSI/ISO C Specification Language, version 1.4, 2009. http://frama-c.cea.fr/acsl.html.

[BFM09c] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for certifying floating-point programs. In Calculemus '09/MKM '09: Proceedings of the 16th Symposium, 8th International Conference. Held as Part of CICM '09 on Intelligent Computer Mathematics, pages 59–74, Berlin, Heidelberg, 2009. Springer-Verlag.

[BO80a] D. Bjoerner and O. N. Oest, editors. Towards a formal description of Ada. Springer-Verlag New York, Inc., New York, NY, USA, 1980.

[BO80b] Dines Bjoerner and Ole Oest. The ddc ada compiler development method. In D. Bjoerner and O. Nest, editors, Towards a Formal Description of Ada, volume 98 of Lecture Notes in Computer Science, pages 1–20. Springer Berlin / Heidelberg, 1980.

[BRS05] Mike Barnett, Rustan, and Wolfram Schulte. The Spec# programming system: An overview, 2005.

[Bur99] Alan Burns. The ravenscar profile. Ada Lett., XIX(4):49–52, 1999.

[CCK06] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006.

[CDH$^+$] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In IN CONF. THEOREM PROVING IN HIGHER ORDER LOGICS (TPHOLS), VOLUME 5674 OF LNCS.

[Cha00] Roderick Chapman. Industrial experience with spark. Ada Lett., XX(4):64–68, 2000.

[CO84] Geert B. Clemmensen and Ole N. Oest. Formal specification and development of an ada compiler - a vdm case study. In ICSE '84: Proceedings of the 7th international conference on Software engineering, pages 430–440, Piscataway, NJ, USA, 1984. IEEE Press.

[Cor10] Microsoft Corporation. Code Contracts user manual, 2010.

[Cro03] David Crocker. Perfect developer: A tool for object-oriented formal specification and refinement. 2003.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM, 18(8):453–457, August 1975.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. J. ACM, 52(3):365–473, 2005.

[dt04] The Coq development team. The Coq proof assistant reference manual. LogiCal Project, 2004. Version 8.0.

[FFS07] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In ESOP'07: Proceedings of the 16th European conference on Programming, pages 173–188, Berlin, Heidelberg, 2007. Springer-Verlag.

[Fil] Jean-Christophe Filliâtre. Why: A multi-language multi-prover verification tool.

[FL10] Manuel Fahndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In Springer Verlag, editor, Proceedings of the Conference on Formal Verification of Object-oriented Software (FoVeOOS 2010), 2010.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In Proc. Sympos. Appl. Math., Vol. XIX, pages 19–32. Amer. Math. Soc., Providence, R.I., 1967.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, CAV, volume 4590 of Lecture Notes in Computer Science, pages 173–177. Springer, 2007.

[GHG$^+$93] J. V. Guttag, J. J. Horning, Withs. J. Garl, K. D. Jones, A. Modet, and J. M. Wing. Larch: Languages and tools for formal specification. In Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[GMP90] D. Guaspari, C. Marceau, and W. Polak. Formal verification of ada programs. IEEE Trans. Softw. Eng., 16(9):1058–1075, 1990.

[Gua89] D. Guaspari. Penelope, an ada verification system. In TRI-Ada '89: Proceedings of the conference on Tri-Ada '89, pages 216–224, New York, NY, USA, 1989. ACM.

[Hed04] Shawn Hedman. A First Course in Logic: An Introduction to Model Theory, Proof Theory, Computability, and Complexity. Oxford Texts in Logic. Oxford University Press, USA, illustrated edition, September 2004.

[Hen90] Matthew Hennessy. The semantics of programming languages: an elementary introduction using structural operational semantics. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[HLL^{+}09] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-01, University of Central Florida, School of EECS, Orlando, FL, March 2009.

[HM94] Peter V. Homeier and David F. Martin. Trustworthy tools for trustworthy programs: A verified verification condition generator. In Thomas F. Melham and Juanito Camilleri, editors, TPHOLs, volume 859 of Lecture Notes in Computer Science, pages 269–284. Springer, 1994.

[HM08] Tony Hoare and Jay Misra. Verified software: Theories, tools, experiments vision of a grand challenge project. In Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions, pages 1–18, Berlin, Heidelberg, 2008. Springer-Verlag.

[HN08] Irfan Hamid and Elie Najm. Operational semantics of ada ravenscar. In Ada-Europe '08: Proceedings of the 13th Ada-Europe international conference on Reliable Software Technologies, pages 44–58, Berlin, Heidelberg, 2008. Springer-Verlag.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576–580, October 1969.

[JES07] Paul B. Jackson, Bill J. Ellis, and Kathleen Sharp. Using SMT solvers to verify high-integrity programs. In AFM '07: Proceedings of the second workshop on Automated formal methods, pages 60–68, New York, NY, USA, 2007. ACM.

[JM97] J. M. Jazequel and B. Meyer. Design by Contract: the lessons of Ariane. Computer, 30(1):129–130, 1997.

[Jon90] Cliff B. Jones. Systematic software development using VDM (2nd ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[Jon03] Simon P. Jones. Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, May 2003.

[JP09] Paul B. Jackson and Grant Olney Passmore. Proving spark verification conditions with smt solvers. Paper regarding the improved results of using SMT-Lib for SPARK., December 2009.

[Kah87] G. Kahn. Natural semantics. In Franz Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, STACS 87, volume 247 of Lecture Notes in Computer Science, pages 22–39. Springer Berlin / Heidelberg, 1987. 10.1007/BFb0039592.

[Kle99] Thomas Kleymann. Hoare logic and auxiliary variables. Formal Asp. Comput., 11(5):541–566, 1999.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design, 1999.

[LCC$^+$05] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. Sci. Comput. Program., 55(1-3):185–208, 2005.

[LM10] K. Rustan M. Leino and Michał Moskal. VACID-0: Verification of Ample Correctness of Invariants of Data-structures, Edition 0. In Proceedings of Tools and Experiments Workshop at VSTTE, 2010.

[LvHKBO87] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. ANNA: a language for annotating Ada programs. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[LW07] K.-K. Lau and Z. Wang. Verified component-based software in SPARK: Experimental results for a missile guidance system. In Proc. 2007 ACM SIGAda Annual International Conference, pages 51–57. ACM, 2007.

[Mey92] Bertrand Meyer. Applying "Design by Contract". Computer, 25(10):40–51, October 1992.

[Mey00] Bertrand Meyer. Object-Oriented Software Construction. Prentice Hall PTR, 2nd edition, March 2000.

[MGL10] Petra Malik, Lindsay Groves, and Clare Lenihan. Translating z to alloy. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, Abstract State Machines, Alloy, B and Z, volume 5977, chapter 28, pages 377–390. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[MMU04] C. Marché, Paulin C. Mohring, and X. Urbain. The krakatoa tool for certification of java/javacard programs annotated in jml. Journal of Logic and Algebraic Programming, 58(1-2):89–106, 2004.

[Mon08] David Monniaux. The pitfalls of verifying floating-point computations. May 2008.

[NN07] H. Riis Nielson and F. Nielson. Semantics with Applications: An Appetizer. Springer, 2007.

[O'N94] Ian O'Neil. The formal semantics of SPARK83. Technical report, Program Verification Limited, 1994.

[ORS92] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, _11th International Conference on Automated Deduction (CADE)_, volume 607 of _Lecture Notes in Artificial Intelligence_, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[Ram89] N. Ramsey. Developing formally verified ada programs. In _IWSSD '89: Proceedings of the 5th international workshop on Software specification and design_, pages 257–265, New York, NY, USA, 1989. ACM.

[Rey79] John C. Reynolds. Reasoning about arrays. _Commun. ACM_, 22(5):290–299, 1979.

[Rey99] John C. Reynolds. _Theories of programming languages_. Cambridge University Press, New York, NY, USA, 1999.

[Rey04] John C. Reynolds. Class notes for CS 819B. `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/www15819Bs2004/` 2004.

[Smi95] Michael K. Smith. Formal dynamic semantics of ava 95. Technical report, 1995.

[SPA08] SPARK Team. _SPARK Examiner: The SPARK Ravenscar Profile_, January 2008.

[Spi89] J. M. Spivey. _The Z notation: a reference manual_. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Sun00] Sun microsystems. _Java Card Techonolgy_, 2000.

[SYK05] Elisabeth A. Strunk, Xiang Yin, and John C. Knight. Echo: a practical approach to formal verification. In _FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems_, pages 44–53, New York, NY, USA, 2005. ACM.

[TDB$^+$07] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, and Pascal Leroy. _Ada 2005 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1 (Lecture Notes in Computer Science)_. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[VR01] Femke Van Raamsdonk. Logic in Computer Science: Modelling and Reasoning about Systems, volume 1. Cambridge University Press, New York, NY, USA, 2001.

[Win93] Glynn Winskel. The formal semantics of programming languages: an introduction. MIT Press, Cambridge, MA, USA, 1993.

[Woo] Jim Woodcock. Tokeneer experiments.

[YKNW08] Xiang Yin, John C. Knight, Elisabeth A. Nguyen, and Westley Weimer. Formal verification by reverse synthesis. In SAFECOMP, pages 305–319, 2008.