A Formal Verification Environment for Use in the Certification of Safety-Related C Programs

von Dennis Walter

Dissertation

zur Erlangung des Grades eines Doktors der
Ingenieurwissenschaften
— Dr.-Ing. —

Vorgelegt im Fachbereich 3 (Mathematik & Informatik)
der Universität Bremen
im September 2010

Datum des Promotionskolloquiums: 16. November 2010

Gutachter: PD Dr. Christoph Lüth (Universität Bremen)
Prof. Dr. Jan Peleska (Universität Bremen)

Zusammenfassung

In dieser Dissertation werden der Entwurf und die Entwicklung einer Umgebung zur durchgängig formalen Verifikation funktionaler Eigenschaften sicherheitsbezogener, in der Programmiersprache C verfasster Software beschrieben. Das Hauptaugenmerk liegt hierbei auf der teils automatisierten und teils vom Benutzer geführten Verifikation nicht-trivialer mathematischer Berechnungen, die etwa im Bereich moderner Sensortechnik eine bedeutende Rolle spielen und deren nachweisliche Korrektheit Voraussetzung einer Zulassung des diese verwendenden Gesamtsystems ist. Zu Beginn der Arbeit wird der gesetzlich-normative Rahmen analysiert, welcher für eine sicherheitstechnische Zulassung von Bedeutung ist, wenn programmierbare elektronische Systeme und damit insbesondere auch Software verwendet werden. Die Schwerpunkte der Arbeit liegen erstens in der Verbindung zweier bestehender formaler Sprachen, nämlich der Logik höherer Ordnung wie sie im Theorembeweiser Isabelle formalisiert wurde, sowie einer programmiersprachenähnlichen Spezifikationssprache im Stile des Design by Contract Paradigmas. Diese Verbindung ermöglicht es, gleichzeitig codenah und auf einer angemessenen Abstraktionsebene funktionale Eigenschaften von Programmen zu spezifizieren. Zweitens wird ein Speichermodell für die Programmiersprache C im Theorembeweiser Isabelle formalisiert, welches hinreichend detailliert ist, um gängige systemnahe Operationen modellieren zu können, während es den damit üblicherweise einhergehenden Verifikationsaufwand in erträglichen Grenzen hält. Schließlich werden auch die Semantik einer Sprachteilmenge von C formal definiert sowie Beweisregeln im Stile des Hoare-Kalküls daraus abgeleitet, so dass ein lückenloser Nachweis der Korrektheit von Programmen bezüglich der spezifizierten Eigenschaften ermöglicht wird. Die Tauglichkeit des Ansatzes wird nachgewiesen, indem die Anwendung in einem realen Projekt beschrieben wird. Konkret wurde mithilfe der Verifikationsumgebung ein substanzieller Teil der normativ geforderten Verifikationsmaßnahmen für eine sicherheitsbezogene Software abgedeckt. Sowohl diese Software als auch die hier beschriebene Verifikationsumgebung wurden von einer zuständigen Stelle bezüglich ihres Einsatzes in sicherheitsbezogenen Szenarien positiv begutachtet.

Abstract

In this thesis the design and the development of an environment for the formal verification of functional properties of safety-related software written in the programming language C is described. The main focus lies on the partly automated and partly user-guided verification of non-trivial mathematical computations that play an important role in modern sensor technology, among others. The demonstration of the correctness of these computations is an absolute prerequisite for their approval by a certification authority. The relevant legal and normative setting is analysed at the beginning of this work, which applies to safety-related systems that make use of programmable electronic devices and software in particular. There are three major achievements contained in this work. Firstly, a formal connection is established between the expressive language of higher-order logic, as it is formalised in the theorem prover Isabelle, and a specification language which stays close to the syntax and semantics of the programming language and which lies in the tradition of the design by contract methodology. This connection allows to specify functional properties of programs in a code-centric and at the same time appropriately abstract fashion, so that the clean mathematical character of higher-level specifications is retained. Secondly, a memory model for the programming language C is formalised within the theorem prover Isabelle which is sufficiently detailed to model low-level memory operations while still keeping the entailed verification overhead in tolerable bounds. Finally, a denotational semantics for a subset of C is formally defined and a Hoare style proof calculus is derived from it such that the correctness of programs with respect to their specifications can be shown in one integrated framework. The applicability of the approach is demonstrated by describing its use in a real project. Concretely, a substantial number of the verification measures stipulated by an international safety standard for safety-related software is covered by employing the verification environment described here. The compliance of both the safety-related software developed in the project as well as the verification environment itself have been officially asserted by a responsible certification authority.

Danksagung

Ich bedanke mich sehr herzlich bei meinem Betreuer Christoph Lüth, der mir die Arbeit an dieser Dissertation ermöglicht und mich stets mit Rat und Tat unterstützt hat.

Weiterer Dank gebührt den ehemaligen Mitgliedern des SAMS-Projekts, allen voran Holger Täubig und Udo Frese, für die gute Zusammenarbeit und etliche Diskussionen, die zur Verbesserung der hier vorgestellten Arbeit beitrugen.

Schließlich danke ich meinen Eltern und meiner Freundin Maren dafür, dass sie nie an mir gezweifelt haben.

Personne n'est sujet à plus de fautes
que ceux qui n'agissent que par réflexion.

Luc de Clapiers, Marquis de Vauvenargues (1715–1747)

Contents

List of Figures

Chapter 1

Introduction

This thesis contributes to the field of software verification. It describes the concepts, the realisation and the application of a methodology to specify desired properties about software systems written in the programming language C and to verify that these properties are actually satisfied. An integral part of this work is the implementation of this methodology in a *formal verification environment*, where by formal both its mathematical foundations and the rigour applied during its construction and use are referred to.

1.1 Motivation

A safety-related system is a system whose malfunction leads to hazardous conditions and therefore puts the lives and health of humans or the integrity of the environment at risk. Computer-based safety-related systems are ubiquitous nowadays, mostly appearing as *embedded systems* inside larger systems. We find them in our cars in the shape of electronic stability control and anti-lock braking systems. Modern aviation and railway transportation are inconceivable without the aid of computer technology. We also rely on them in X-ray computed tomography, lung ventilators, cardiac pacemakers and several other medical equipment. Another example is the industrial process and automation sector, where an increasing number of tasks is performed by computer-controlled machines, but where an interaction between machines and personnel still takes place.

Software safety. Due to the increasing complexity of these systems, which rises with the number of requirements on their functionality, more and more safety-related functionality is realised through software. This makes software safety an important aspect of the overall system safety warranting its consideration in all applicable modern safety standards, such as DO-178B, IEC 61508 or EN ISO 13849 [134, 80, 52]. It is commonly defined as the freedom from hazardous conditions or the absence of unjustifiable risks in a software system. Software safety can only be achieved through a combination of several measures, including detailed planning of the software life cycle, proper assignment of responsibilities, the establishment of tried and tested programming guidelines, and the management of software configurations. All these are arguably ‘soft’ measures that influence software quality indirectly. The primary means to achieve software safety is in any case to show that the created software satisfies the requirements laid upon it. This necessitates the *specification* of requirements and the carrying out of *verification* activities. Safety-related software development processes distinguish several requirement levels that correspond to the different levels of abstraction at which the software under development is viewed. They assign verification activities to the artefacts produced at each level as well as to the transition steps from one level to the next, more refined one. Ensuring *traceability* is a key goal, so that low-level requirements, designs or software functions are always derived from higher level requirements. This validates the completeness of the top-level specifications and avoids the introduction of unwanted additional functionality.

On the source code level the verification measures are intended to verify compliance with the low-level requirements. The measures can be classified into *testing*, *static analysis* and *formal methods*. The objective of testing is to discover defects in the software and to provide exemplary evidence of the compliance with specific low-level requirements by running the software with defined input vectors. It is the only means able to eventually validate the correctness of hardware/software integration, since this always involves the execution of the software on a specific platform. Static analysis comprises automatic techniques used to ensure a well-defined and restricted set of non-functional properties of the software, such as type correctness, the exclusive use of previously initialised variables, or the observance of syntax-based coding guidelines. Their application is advisable, but the delivered guarantees are rather weak and can hence only support a more comprehensive verification by other means. Formal methods embody the techniques that can deliver the strongest guarantees about the correctness of a software, since both the models they work on as well as the analyses they perform are by definition equipped with a mathematically precise meaning. They are commonly divided into *model checking*, *abstract interpretation* and *deductive methods*. The former two have recently gained much attention through their application in industrial projects [48, 138] and their discussion in a supplement of the latest revision of the most relevant software safety standard in aviation, DO-178C [31]. Their most intriguing feature is that they generally provide a large degree of automation, while at the same time allowing the verification of well-defined yet interesting properties of a software. Examples include verifying the bounds on the execution time or the stack usage of extremely large ($> 100 \cdot 10^3$ lines of code) software systems.

Deductive formal methods. In the context of software verification deductive methods are verification methods that are based on formal logic. Formal models are expressed in terms of definitions and axioms of a suitable logic, and properties about these models are established by proving theorems as derivations according to the logic’s rules of inference. Well-known examples of formal methods used on higher abstraction levels include CSP or the Z specification language and their associated reasoning tools. More recent offspring of these are the π-calculus and the Alloy modelling language. The standard deduction-based technique for specifying and verifying low-level *functional requirements* about source code is the use of mechanised Hoare logics. Here, mechanised refers to their realisation in software as opposed to mere pen and paper formalisations. They are characterised by the use of a formal language —most commonly some variant of first-order logic— to precisely state required properties of program units. These are split into preconditions and postconditions, expressing assumptions about the program unit's environment before the unit is executed and guarantees about it after execution, respectively. A statement asserting that a program satisfies its specification is generally written as a *triple* of the precondition, the program unit and the postcondition. For example, the triple

$$\vdash \{\ \mathit{True}\ \} \quad \textbf{if}\ (\mathsf{x} < 0)\ \mathsf{x} = -\mathsf{x}; \quad \{\ \mathsf{x} \geq 0\ \} \tag{1.1}$$

asserts that after execution of the displayed program unit under no assumptions (more precisely: if *True* holds) the variable x will be non-negative.

Hoare logics are equipped with a simple deductive system. This system is used to decompose triples and to obtain a collection of purely logical propositions in the formal language. The logical validity of these propositions implies the correctness of the program. Classical examples are the rule for the conditional

$$\frac{\vdash \{P \wedge b\}\ c_1\ \{Q\} \qquad \vdash \{P \wedge \neg b\}\ c_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2\ \{Q\}} \tag{1.2}$$

and the rule to logically strengthen the precondition and to weaken the postcondition,

$$\frac{P' \longrightarrow P \qquad \vdash \{P\}\ c\ \{Q\} \qquad Q \longrightarrow Q'}{\vdash \{P'\}\ c\ \{Q'\}}. \tag{1.3}$$

Interpreted operationally from the bottom to the top, this rule yields two purely logical propositions (the implications) and allows to transform the triple's precondition and postcondition such that further rules become syntactically applicable. The validity of the propositions is proven in the logic of the formal specification language and practically involves the use of automatic or interactive theorem provers.

The class of properties that can be expressed in this way is determined by the definition of correctness, or validity of a triple, and by the expressivity of the formal language in which the specifications are formulated. In any case, from an abstract viewpoint classical Hoare logics regard programs as functions $\mathcal{I} \times \Sigma \to \mathcal{P}(\mathcal{O} \times \Sigma)$ from some input $I \in \mathcal{I}$ and an environment $\sigma \in \Sigma$ (which might consist of the program's global variables, open file descriptors, or the contents of a hard disk) to a collection of results consisting of the function's output $O \in \mathcal{O}$ and a possibly modified environment $\sigma' \in \Sigma$. This is a very general approach regarding the functional properties that can be expressed. On the other hand, it ignores all properties that pertain to the execution steps (or *trace*) that constitute a program run. Examples of properties that cannot be reasoned about in Hoare logics are a program's execution time, its stack usage or the compliance with the usage protocols of libraries. Extensions that encode these properties to some degree exist, however [8].

The focus on functional properties is a clear separation of concerns and their sublime importance for software safety is undisputed. However, a major critique of Hoare logics, and deductive methods in general for that matter, is that their application is usually not entirely automatic, but demands extensive expertise and manual interaction from their users. Moreover, the practical relevance of software verification with Hoare logics is limited by the fact that it does not harmonise well with language constructs present in practical programming

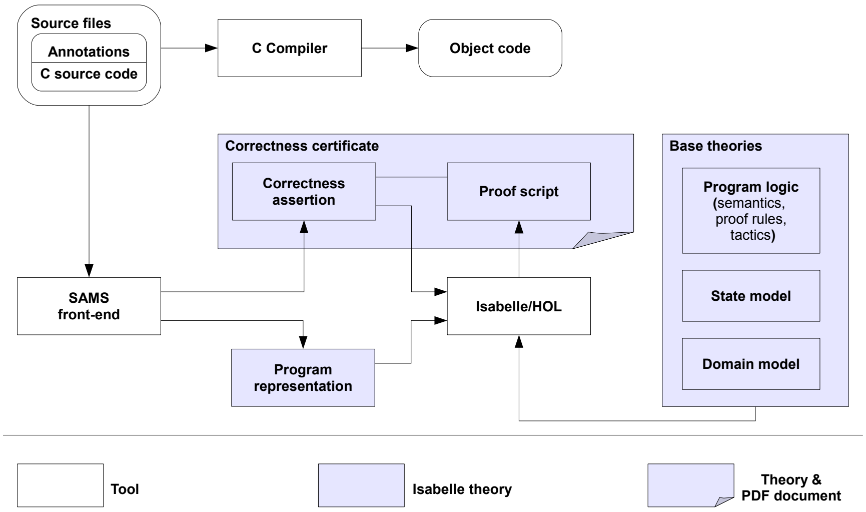


Figure 1.1: Architectural overview of the SAMS verification environment

languages. The most prominent examples are memory address arithmetic and pointer aliasing, which are ubiquitous in programs written in C.

The interest in Hoare logics and related techniques has increased recently due to a series of success stories related to their (academic) application to verify interesting algorithms such as memory garbage collectors. The degree of automation could be improved due to the progress made in the reasoning power of interactive proof assistants and automatic theorem provers. Moreover, the mismatch between Hoare logics and C's low-level view on memory was remedied by devising interesting new formalisations of memory models that avoid certain problems such as excessive aliasing by their very construction.[1]

Overview of the methodology. The application domain targeted in our work is that of software for safety-related control systems and industrial robotics. High-level requirements in these domains are especially well-suited for a mathematically precise formulation. A good example is the safety function of ensuring collision avoidance for mobile robots. The distance between a moving robot and the objects in its surrounding that is required to ensure collision avoidance can be expressed as a function over the robot's characteristics such as its speed, braking behaviour and shape. Another example is the monitoring of objects transported on band-conveyors: a laser scanner that monitors the band's surface might be required to raise a halt signal whenever an object deviating from specified geometric characteristics is detected. A high-level requirement might define these characteristics in terms of properties such as having right-angled boundaries, a planar surface or a maximum height.

The approach to software verification pursued in this thesis lies in the tradition of Hoare logics. Its basic architecture and the verification work flow are

[1] References for this paragraph can be found in Sec. 1.4 discussing related work.

depicted in Fig. 1.1. High-level requirements and concepts of the application domain in which the software system to be verified is used are formalised as theories of the proof assistant Isabelle [116]. This is referred to as the *domain model.* The model that was developed for the concrete verification effort described in this thesis is based on two-dimensional geometry. It formalises concepts such as convex hulls of point sets, approximations of circular arcs by polygonal lines, or several standard vector operations. The *C source code* is then *annotated* with specifications expressing its required behaviour. A characteristic feature of our methodology is that this behaviour is formulated at the level of the domain model. This is achieved by abstracting program entities to domain entities and by referencing entities defined in the domain model in code specifications. For example, a source code function computing a robot's braking trajectory is specified abstractly by interpreting the robot as a two-dimensional object and expressing its motion in terms of mathematical geometric transformation functions, and *not* by directly stating properties of the data structures that merely represent the robot's shape or the parameters of the transformation function in the code. Specifications use the abstraction facilities of the *state model* —i. e., the formal representation of the program's environment— to formally associate program entities with their high-level domain counterparts.

Technically, the annotations are hidden from the parser of a regular compiler by embedding them in specially formatted code comments. This ensures that the verified code can directly be translated to object code without modifications. For verification, the annotated source is parsed by a front-end which does recognise the specifications. It type checks specifications as well as source code and ensures the observance of coding guidelines. The tool eventually translates its input to an abstract syntax tree (called the *program representation*) in a format that is understood by the theorem prover. Additionally, a *correctness assertion* is output; this is a theorem whose proof establishes the correctness of the program unit under consideration with respect to its annotated specification. Verification is eventually reduced to theorem proving. The theorem finally has to be proven by a human verifier within the interactive the theorem prover Isabelle/HOL, utilising the *program logic* which includes procedures that automate large parts of the proof work as well as high-level theorems from the domain model.

Statement of the thesis. The hypothesis that is made and investigated in this thesis is that formal functional verification is an adequate means to increase the confidence in software employing sophisticated mathematical computations to such an extent that the software is ready for use in safety-related systems. More specifically, we argue first that safety requirements can be formulated at the source code level in an unambiguous, well-defined language that allows to maintain a rigorous connection to a formal model of the high-level safety requirements. Second, we claim that it is feasible to build precise formal models of a relevant class of programs written in the C programming language, which can be reviewed by external bodies such as a certification authority. And third we assert that modern interactive theorem provers for higher-order logic are appropriate verification vehicles by which the compliance of the program model with the formalised requirements can be established.

1.2 The SAMS Project

The work that this thesis is built upon was done in the context the SAMS project, which is an acronym for ***S**afety Component for **A**utonomous **M**obile **S**ervice Robots*.[2] The project was a cooperation between Leuze electronic[3], a manufacturer of sensor equipment and specifically laser scanners, the Mathematics and Computer Science Department of the University of Bremen, and the Safe and Secure Cognitive Systems research department of the German Research Center for Artificial Intelligence (DFKI)[4]. The goal of the SAMS project was the development and certification of a safety component for autonomous mobile robots. The task of this component, which is supposed to be mounted on an *equipment under control* (EUC; here: the robot), is to provide collision avoidance between the moving EUC and its environment by sensing the environment with an associated safety laser scanner, by evaluating the scanner's distance measurements to detect possibly hazardous obstacles, and by timely emitting an emergency stop signal to bring the EUC to a halt. In contrast to existing industrial approaches to collision avoidance, the decision whether to emit the emergency stop signal also depends on the current velocity of the EUC: at slow speed obstacles may be detected nearer to the EUC without intervention than at higher speed.

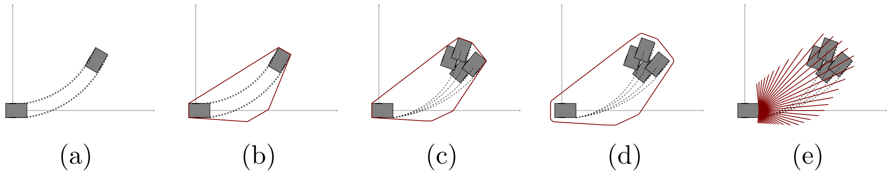

(a) (b) (c) (d) (e)

Figure 1.2: Computation of safety zones

The main task of the software of this component is to compute a conservative over-approximation of the actual *braking area* covered by the EUC during braking, given its current forward and angular velocity. This approximation is called a *safety zone*. Fig. 1.2 depicts their algorithmic computation: (a) The motion of the EUC is anticipated according to a simple braking model, (b) the covered area is approximated by a convex polygon, (c) the uncertainty of the velocity measurements as well as (d) further safety margins related to the response time of the overall system and to numerical imprecision are incorporated, and (e) the safety zone is eventually transformed into a representation that allows a fast comparison with the measurements of the laser scanner to detect obstacles, where a minimum distance is assigned to each laser beam emitted by the scanner. Slightly oversimplifying, an obstacle is detected whenever the measurements of the laser scanner yield smaller values than those required by the safety zone representation. Whenever this happens, the emergency stop signal is emitted. Solutions like this are well-known in the scientific robotics community [93, 60, 108]. The novel aspect of the SAMS project is that such a

[2] The project was funded by the German Federal Ministry of Education and Research (FKZ 01 IM F02 A)

[3] `http://www.leuze.de`

[4] `http://www.dfki.de/web/research/sks`

software has been certified by an authority (TÜV Süd Rail GmbH) as conforming to the requirements of the international safety standard IEC 61508-3 [80]. This substantially aids in providing a safety case for safety-related systems that fall under the scope of the EC machinery directive (2006/42/EC), but also in other scenarios in which the conformance with regulations is assumed or at least supported by the conformance with IEC 61508.

1.3 Contributions

This work contributes to the field of software verification in the following ways:

1. We provide a formalised memory model for C programs that combines the properties of the Burstall-Bornat split heap approach [34, 29] with the unrestricted use of C's address-of operator &. In particular, the model allows to treat the addresses of local variables and structure members as first-class citizens that can be assigned to pointer variables, passed as function arguments, etc., while still keeping the induced verification overhead in tolerable bounds.

2. We developed a specification language for C programs which allows to establish a formal connection between high-level concepts of the application domain and their concrete representations as C datatypes. The language is a hybrid between traditional first-order design-by-contract specification languages such as ACSL [18] and the expressive language of Isabelle/HOL. The language allows us to express the properties of program functions in terms of domain concepts, rather than in terms of program variables. This gain in abstraction significantly increases the value of code specifications and the confidence in the correctness of the specification w. r. t. higher-level requirements.

3. We created a verification environment in which all constituent parts have been developed with formal scrutiny within the theorem prover Isabelle, except for a programmatic front-end that translates annotated C source code to the prover's input language. All associated theories define conservative extensions of the default Isabelle/HOL theory library. The development comprises the memory model for C, a deep embedding of C programs in Isabelle/HOL together with an intuitive denotational semantics, a model of parts of two-dimensional geometry together with the theorems necessary to prove concrete programs correct, a set of proof rules used to derive verification conditions and finally tactics that aid in proving the generated verification conditions. While similar work has been done by Schirmer [136], our approach is different in that it more clearly separates the specification and verification phases by attaching specifications to source code instead of their representations in the theorem prover. Moreover, our scope is more limited, since the design of the environment was guided by the concrete needs of verifying safety-related robotics software. We claim that this deliberate restriction simplifies the tool qualification with responsible authorities.

4. We applied the verification environment in a real-world industrial project, where we proved the correctness of a software module used to achieve

collision avoidance for an automated guided vehicle protected by safety equipment, in particular a laser scanner. A letter of conformance by the certification authority TÜV Süd Rail GmbH attests us the applicability of our approach for software used in safety contexts up to SIL 3 of the safety standard IEC 61508.

1.4 Related Work

Previous work that is directly related to the topic of a particular chapter in this thesis is discussed at the beginning of the respective chapter. A classification of our specification language is given at the beginning of Chapter 3. The discussion of other formalisations of memory models and the semantics of C is provided in Chapters 4 and 5, respectively. Our approach to verification is not fully automatic, but we rely on user interaction to discharge proof obligations that cannot be handled by the automatic procedures built into Isabelle. The relation between our approach and fully automatic logic-based verification condition generators is explained in Chapter 6.

The more generally related work falls into the three categories of Hoare logics for program verification, automatic techniques for the analysis of software, and applications of formal verification in a certification context.

Hoare logics. Hoare logics have a long tradition in computer science, originating from the seminal papers by Floyd [59] and Hoare [76]. This work has been extended to incorporate language features such as procedures, recursion, memory allocation and arrays, among others (e. g., [35, 34, 91]). More recently, Bornat [29] and Nipkow [115] have given formalisations of Hoare logics for pointer programs in the theorem provers Jape and Isabelle. The former includes a proof of the Schorr-Waite graph marking algorithm, which has since become a benchmark for tools concerned with proving programs over linked data structures correct. Mehta and Nipkow [105] transfer this proof to Isabelle/HOL.

There exist many tools for the interactive verification of Java programs based on Hoare-style pre-/postconditions (e. g., [10, 20, 102, 3]), but there are less such tools for C, probably due to its low-level nature and under-definedness. The two most advanced tools are probably Why [55] (together with its instantiation to C named Caduceus [56]) and the framework developed by Schirmer [136] which is also used within the Verisoft project [5]. Our verification environment is close in spirit to these two tools. Caduceus transforms concrete C code to the intermediate language of Why, which excludes aliasing between variables. This enforces a rather involved encoding of C's program variables and is a programmatic step whose correctness must be trusted. Schirmer, on the other hand, also differentiates between C programs and a simpler program representation called Simpl, but proves the equivalence between the two formalised semantics, hence increasing trustworthiness. Schirmer and Wenzel [137] extend on this work by defining a parametric memory model which makes use of Isabelle's concept of local theories. Gast [62] is also concerned with efficient representations of memory to enable programmer-friendly program verification. His concept of memory layouts allows to isolate the structural memory assertions of separation logic [132] from the logical essence of specifications while retaining several properties useful to restrict aliasing between linked data structures.

SPARK [12] is a subset of the Ada programming language used in critical systems and comes with its own verifier that allows to prove programs correct w. r. t. manually annotated pre-/postconditions.

Recently, the interactive formal verification of systems code – i. e., programs working at the lowest level of the software stack such as operating systems, garbage collectors, or firmware – has gained much attention. Klein [88] gives an overview of the advances made in formal operating systems verification. The verification of the seL4 microkernel is described in [89]. Leinenbach and Petrova [95] verified a simple compiler for the C0 subset of C.

Automatic verification techniques. It is obviously extremely appealing to be able to prove properties of programs correct without human intervention. The price to pay in virtually all cases is a constraint on the kind of properties that can practically be verified.

Blanchet et al. [25] describe Astrée, a tool based on abstract interpretation [49] that automatically proves the absence of run-time errors in synchronous, time-triggered, safety critical C programs, which include undefined behaviour such as indexing arrays out of bounds, integer overflow and division by zero. The tool as been successfully applied in industrial settings.

Motivated by the success of hardware model checking, software model checkers have become popular recently. These tools are also used to verify a fixed set of program safety properties, but also allow to verify simple quantifier-free assertions inserted into the code by the developer. Their application has been most successful in the verification of device drivers and API usage protocols; SLAM [9] and BLAST [75] are two examples of this. Clarke et al. [41] apply bounded model checking to verify C programs. Essentially, the bound applies to the maximum number of loop iterations considered by the tool. At the cost of making the verification fail when no safe upper bound can be determined, their tool can be applied as a debugger, because concrete error traces can be produced and visualised if property violations are detected. Their CBMC tool has been evaluated in several case studies, e. g., [87, 130]. Chaki et al. [36] present a tool for the automatic verification of C programs exhibiting message-passing concurrency. Both programs and specifications are modelled as labelled transition systems (LTS) and a model checking technique based on counter-example guided abstraction refinement [40] is employed to prove trace containment between the two LTS. Schlich and Kowalewski [138] evaluate the use of software model checking in embedded systems. They point out that industrial users miss the possibility to analyse arithmetic and continuous properties of their systems when applying model checking. An excellent survey on automatic formal software verification is given by D'Silva et al. [53].

A recent tool for the verification of operating systems code written in C is VCC [43]. It requires manually supplied annotations of loop invariants and function contracts, but uses the SMT solver Z3 [51] to automatically discharge the generated proof obligations. The tool has been used to verify mainly structural properties like data type invariants of concurrent C programs [45].

A technique for (almost) automatically proving functional properties of Java programs is described by Zee et al. [153]. They prove the functional correctness of linked data structure implementations by combining a collection of decision procedures, first-order theorem provers and interactive theorem proving in higher-order logic. They rely on invariants and assertions that must be supplied manually, but otherwise achieve a high degree of automation by splitting verification conditions into large conjunctions in which each conjunct can be passed to an appropriate prover. They use abstract concepts like sets and relations in specifications at the interface level, similar to what is achieved by our representation functions.

Certifying software using formal verification. The benefits of using formal functional verification in a certification context, an overview of the approach described in this thesis as well as the experiences made during verification and certification are also described in the papers [61, 100, 150]. A major aspect by which this work differs from many others also concerned with formal software verification (e. g., [4, 95, 98, 105]) is that it is performed in a real world project in which the advocated methods and the software thus verified are actually examined by a certifying authority. We share this characteristic with Heitmeyer et al. [74], who describe the verification of security properties for a memory separation kernel used in an embedded device. Their method is based on refinement [2] and the focus is on establishing a formal connection between a top-level specification expressed in terms of a state machine model and annotated C source code. In contrast to our work, the code annotations are treated as assertions whose correctness is only manually validated.

O'Halloran [119] discusses the high cost incurred by the requirement of independence between software development and verification which is set up by aviation standards like DO-178B [134]. He acknowledges the usefulness of automatic code generation from more abstract models and advocates a formal approach to verifying the correctness of the output of the code generator. This is done by a mechanical translation of both the input model and the output source code to a dialect of Z and a subsequent refinement proof.

Barnes et al. [11] present a case study demonstrating how formal verification can be used to achieve conformance with the upper (more demanding) levels of the Common Criteria, an international agreement forming a basis for evaluating the security of information technology products. They implemented an access control software prototype in a subset of Ada and formally proved both that the code satisfies its design specification as well as that the design specification is a refinement of a more abstract top-level specification. They conclude that formal verification helps in certifying high-integrity software in a cost-effective manner.

Basir et al. [16] verify automatically generated code using theorem proving in a certification context in the aerospace domain to enable the use of untrusted code generators, relying on automatic first-order provers. The approach is extended in [17] to derive safety cases, i. e., structured arguments about system safety, that reflect the hierarchical structure of the models from which the software is automatically generated.

Peleska [126] and Löding and Peleska [97] integrate testing and formal approaches, using abstract interpretation and model checking, where abstract interpretation is used to prove the absence of run-time errors and to efficiently exclude infeasible paths, speeding up the constraint solving process used to generate input vectors for reachability testing. Their approach is compliant with applicable software safety standards in avionics and railway domains. They advocate a unit-at-a-time verification paradigm supported by the expertise of specialists; code specifications in terms of quantifier-free pre-/postconditions are therefore encouraged for functional testing.

Finally, an overview over the aspects to be considered by authorities when formal methods are used in the certification of software is given by Rushby [135].

1.5 Overview

This thesis is structured as follows: Chapter 2 sets the scene by providing background on regulations applicable when certifying robotics software and by outlining the most important aspects of the theorem prover Isabelle. In Chapter 3 we present the specification language developed for specifying deep functional properties of C programs. Chapter 4 contains a presentation of the memory model formalisation on which the semantics of a subset of the C programming language are based, as explained in Chapter 5. The Hoare-style proof rules used for the generation of verification conditions are given in Chapter 6, together with an account of the algorithm for solving aliasing constraints. The application of the overall verification environment to the code developed in the SAMS project as well as some reflections of this effort can be found in Chapter 7. We conclude this thesis in Chapter 8.

Chapter 2

Legal and Technological Background

This chapter provides the relevant background information necessary for an understanding of the rest of the thesis. On the regulatory side it discusses the EC machinery directive and the international safety standard IEC 61508. On the technological side it provides the necessary conceptual information about the theorem prover Isabelle and introduces its notation, which is used in several parts of the thesis.

2.1 Laws, Standards and Guidelines

In this section a brief introduction to the legal conditions that make a verification effort for safety-related software a necessity is given. The verification environment described in this thesis can be seen as one solution to (partially) fulfil this obligation. Since legal conditions differ substantially from country to country, only the situation in the European Union is considered.

We assume a setting in which an automated guided vehicle (AGV) is to be used in an industrial storehouse to transport packages from a packaging station to a dispatch station. The packages are put on the AGV and removed from it either automatically or manually, but the important fact is that the AGV's movement from station to station does not require human intervention.

Law At this point we have already entered legal grounds: AGVs are machinery according to the definition of the European Parliament and the Council [54][1], and hence the provisions of this directive apply to them. A primary goal of this directive is to state and enforce health and safety requirements relating to the design, construction and placing on the market of machinery. Part of these provisions is the carrying out of a risk assessment *"to determine the health and safety requirements which apply to the machinery"*[54, Annex I, General Principles]. This *risk assessment* might partially proceed as follows:

[1] or rather national law or regulations implementing the directive (e.g. the German *Maschinenverordnung* or the *Machinery Regulations* in the UK)

EN 1525	Safety of industrial trucks – Driverless trucks and their systems
EN ISO 3691-4	Industrial trucks – Safety requirements and verification – Part 4: Driverless industrial trucks and their systems
EN 1175-[1-3]	Safety of industrial trucks – Electrical requirements
EN ISO 12100-1	Safety of machinery – Basic concepts, general principles for design – Part 1: Basic terminology, methodology
EN ISO 13849-1	Safety of machinery – Safety-related parts of control systems – Part 1: General principles for design
EN ISO 13849-2	Safety of machinery – Safety-related parts of control systems – Part 2: Validation
EN 61496	Safety of machinery – Electro-sensitive protective equipment – Part 1: General requirements and tests
EN 61508-[1-7]	Functional safety of electric/electronic/programmable electronic safety-related systems

Figure 2.1: Standards relevant for the development of AGVs and safety laser scanners

1. During the *identification of hazards* it becomes evident that the AGV's driveway might interfere with the workspace of personnel.

2. A *risk estimation* shows the risk and in particular the non-negligible probability of serious injury for personnel through the AGV.

3. To reduce this risk, a *protective measure* must be taken. Because the interference between personnel and AGV cannot be avoided, it is decided to install a device to detect the presence of persons. This device can then be used for collision avoidance. The concrete device opted for is a laser scanner.

The laser scanner, a *safety component*, must itself be considered machinery according to the directive, so that its construction and placing on the market are subject to the same requirements.

Standards It is important to note that the directive itself is not the document according to which the development of machinery is practically done. Due to its wide application area, ranging from sawing machinery over compression machinery to logic units ensuring safety functions, it is necessarily too general to be used as a reference for proving compliance with the requirements stated therein. The directive acknowledges this in its whereas clause (18): *"In order to help manufacturers to prove conformity to these essential requirements, and to allow inspection of conformity to the essential requirements, it is desirable to have standards that are harmonised at Community level for the prevention of risks arising out of the design and construction of machinery."* Practically this means that the proof of compliance with the provisions of the directive is done or at least supported by the proper application of domain-specific standards. Fig. 2.1 represents an incomplete list of standards relevant to the development of said AGV and laser scanner. Standards relevant to AGVs have been taken from [147].

Approaching our domain of interest (software verification) further, we note that the laser scanner will perform a *safety function* intended as a risk reduction measure for the operation of the AGV. It is also clear that a laser scanner, containing microprocessors and running software for the interpretation and filtering of its sensor data, is a *programmable electronic device*. Therefore, among others, IEC 61508 – Functional safety of electric/electronic/programmable electronic safety-related systems (which in the European Union has been published as EN 61508) is an applicable standard. It covers the relevant aspects to be considered when electric or (programmable) electronic systems are used to execute safety functions. Even though a manufacturer is not legally obligated to apply the standard, there are good reasons why this should be done, as explained in Sec. 2.1.2. But even if it were not applied, the following reasoning would be valid, because of the directive's provision of taking the *state of the art* into account when placing machinery on the market. The state of the art, which has no formal definition within the directive, is essentially defined by applicable current standards, like IEC 61508.

IEC 61508 states requirements on each phase of the system life cycle. This concerns technical aspects of a system as well as process-related aspects. In particular, and in contrast to many previous standards, it states elaborate requirements on the software employed in a safety-related system. A whole part of the standard (IEC 61508-3) is entitled "Software requirements" and accounts for 52 pages alone. Major sections thereof are concerned with the verification and validation of software, covering its planning, execution and documentation.

At this point it has hopefully become clear that software verification does not only have the "voluntary" objectives of making the software more reliable, making software development more productive and feeding consultants and researchers involved in verification. Rather, the need for software verification for safety-related systems can be traced back to legal regulations. This has been taken into account during the development and application of the verification environment.

Guidelines One further kind of official document was of importance in the design of the verification environment. These are the *Guidelines for the use of the C language in critical systems (MISRA-C guidelines)* [109]. The use of guidelines is quite natural for any project. Generally, guidelines may relate to best-practices for particular tasks, enforce uniform procedures, or define how certain software tools, documents or machines are to be used. Obviously, the term has a very broad meaning; the MISRA-C guidelines main objective is to define a subset of the C language —often referred to as MISRA-C, though not in the guidelines themselves— that is considered appropriate especially for use in safety-related systems.[2] Their application is, again, not specifically mandated by any standard or law; the use of a language subset, however, is encouraged by IEC 61508-3 and is almost inevitable when dealing with an under-defined language like C.

After this motivating fast-forward introduction, a closer look is taken at the EC machinery directive and the safety standard IEC 61508, since they played an important role in the SAMS project and eventually had an influence on the design of the verification environment.

[2] The term "critical" used in the guidelines' title is more general and encompasses more than safety.

2.1.1 EC Machinery Directive

The *Directive 2006/42/EC of the European Parliament and of the Council of 17 May 2006 on machinery, and amending Directive 95/16/EC (recast)*, colloquially abbreviated as *machinery directive*, is a European directive setting up requirements on the design, construction, placing on the market and putting into use of machinery. Its main goals are to remove trade barriers and to enable free movement of goods through a harmonisation of law among the member states of the EU, and to ensure the health and increasing the safety of persons exposed to machinery. The latter goal is pursued by defining health and safety requirements relating to the design and construction of machinery.

Scope To understand the scope of the directive, a look at the definition of the term "machinery" is in order. This definition is given in Article 2 of the directive as *"an assembly, fitted with or intended to be fitted with a drive system other than directly applied human or animal effort, consisting of linked parts or components, at least one of which moves, and which are joined together for a specific application"*. Further variations are listed to also capture assemblies not fully satisfying this definition. In any case, the main characteristics of a machinery are the presence of a *drive system* and of *movable parts*. In addition to machinery, the scope (Article 1) also encompasses *safety components*. In indicative (i. e. incomplete) list of safety components is given in Annex V, with the most relevant items in the context of this thesis being *"logic units to ensure safety functions"* and *"emergency stop devices"*. The former has prominently been added to the list during the most recent amendment; it was not present in the previous machinery directive 1998/37/EC, which was in force until December 2009.

Structure The machinery directive is structured into three parts: First, a sequence of so-called whereas clauses lists the most important considerations that led to the adoption of the directive as well as its concrete shape. These clauses are not legally binding, but rather of explanatory character. Second come 29 articles[3] that make up the main body of the directive. These articles refer to 12 subsequent annexes containing more detailed information about particular subjects, e.g. the health and safety requirements or a procedure for compiling a technical file.

Notable considerations The considerations, or whereas clauses, are not of direct relevance here; for instance they contain discussions about particular aspects of why or why not the directive applies to certain machinery. Two clauses, however, are noteworthy in this context: (14) points out the necessity of satisfying the essential health and safety requirements of Annex I, but relaxes this by noting that the state of the art as well as technical and economical requirements should be taken into account in achieving this aim. While in its wording this is a limiting statement —opening a loophole for machinery that cannot reasonably be made strictly as safe as required by the directive's provisions— it also gives leeway to the argument that in order to show compliance with the

[3] Being a computer scientist and not a lawyer, we allow ourselves to write all legal terms —article, clause, annex, etc.— in lower case, except when referring to explicit entities, e. g. Annex I

directive one may and should refer to current best practices and standards and apply them. This aspect is relevant for the discussion of why standards like IEC 61508 should be applied in the manufacture of machinery. Another interesting clause (23) demonstrates that the directive follows current trends of going beyond analyses that merely try to identify the possible hazards a machinery might evoke, towards comprehensive risk assessments. A risk assessment, above all, additionally asks for the estimation, evaluation and comprehensive documentation of identified hazards, in terms of their probability and the extent of the possible harm underlying the hazard. The terminology relating to safety will be further explained in Sec. 2.1.2.

Articles The 29 articles form the main part of the directive. In the context of a safety-related system development the most relevant aspects are the definition of the requirements that apply to the placing on the market and putting into service of machinery (Article 5), as well as the stipulation of the procedures for assessing the conformity of machinery (Articles 7 and 12). Other articles cover topics like market surveillance (Article 4), postulating that member states should take measures to withdraw machinery from the market that does not comply with the directive; freedom of movement (Article 6), asking member states not to prohibit or impede the placing on the market and putting into service of compliant machinery; or a clarification on the exact shape of the CE marking (Article 16) that indicates compliance with the directive.

Six provisions are given in Article 5, which every manufacturer has to obey before he is allowed to place machinery on the market or put it into service. They are the following:

1. Machinery must satisfy the relevant essential health and safety requirements of Annex I.
2. A technical file must be assembled for the machinery. This file documents the compliance with the requirements of the directive. It must contain technical drawings of the machinery, a documentation of the risk assessment performed, the standards that have been used, test results, the instructions for the machinery, and the EC declaration of conformity. The details are given in Annex VII. The importance of instructions for the machinery is emphasised by repeatedly requesting them in a provision on its own.
3. Conformity with the directive must be assessed in accordance with Article 12.
4. The EC declaration of conformity, which is presented in Annex II, must be drawn up.
5. The CE marking must be affixed on the machinery.

This puts two main burdens on the manufacturer: he must take measures to satisfy the health and safety requirements, and he must assess the conformity with the directive, i. e. in particular with the health and safety requirements.

Article 12 lays down three possible assessment procedures. Which one of these procedures can or must be applied depends on the type of machinery and on the fact whether or not it was manufactured in accordance with harmonised standards. A *harmonised standard* is a standard with a special status within the directive. Article 7 introduces a *conformity assumption*, under which machinery that was developed according to a harmonised standard is assumed to comply with the health and safety requirements that are covered by that standard.

One has to keep in mind that the directive applies to any kind of machinery, not just those one would call safety-related. Therefore, it is clear that for most machinery on the market a simple and economical assessment procedure is needed. The easiest assessment procedure is self-certification, in which the conformity is certified by the manufacturer itself, rather than by an external certification authority. These so-called *internal checks* are described in Annex VIII and essentially demand that the manufacturer must ensure the availability of the technical file for each representative type of the series, and take *"all measures necessary"* to ensure compliance of the machinery with the technical file and the directive.

Annex IV provides an exhaustive list of machinery where safety is of greater concern. For these machinery self-certification is only possible if harmonised standards have been applied in the manufacturing process and if they cover all relevant health and safety requirements of the directive. Otherwise, conformity must be assessed by an external *notified body*, i. e. a body that has the formal authority to perform such an assessment and certify compliance. There are two options: the manufacturer can perform an EC type-examination (Annex IX) as well as internal checks on the manufacture, or he can install a quality assurance system in his company (e. g. as described by the ISO 9000 series). A notified body must be involved for both the type-examination and the assessment of the quality assurance system. It is hoped by manufacturers that the latter will prove to be a more cost-effective route to compliance [99].

Obviously, the chance to avoid involving an external body makes the use of harmonised standards highly attractive, not least economically. Two common reasons why manufacturers might revert to other standards are that for particular machinery there might be (not yet) a harmonised standard available, or that other standards are more attractive, because they are more relevant outside the EU. However, even non-harmonised standards can practically be used as a supportive argument in EC declarations of conformity [121]. The conditions determining the possible procedures for assessing conformity are depicted in Fig. 2.2.

Annexes The twelve annexes provide the details of the provisions laid down in the articles. They also have legally binding character. Annex I can be regarded as the heart of the directive, containing the essential health and safety requirements. They account for 30 pages, i. e. roughly half of all pages of the directive. Importantly, it is demanded that a risk assessment is carried out to determine the health and safety requirements that apply to the machinery, and that the machinery is then designed and constructed according to the assessment results. The risk assessment must identify the limits of the machinery and the hazards that it generates. By estimating and evaluating the risks, the manufacturer must further determine whether the remaining risk is acceptable and in accordance with the directive, or whether more risks need to be eliminated or reduced. The highest priority in this respect is given to the safety integra-

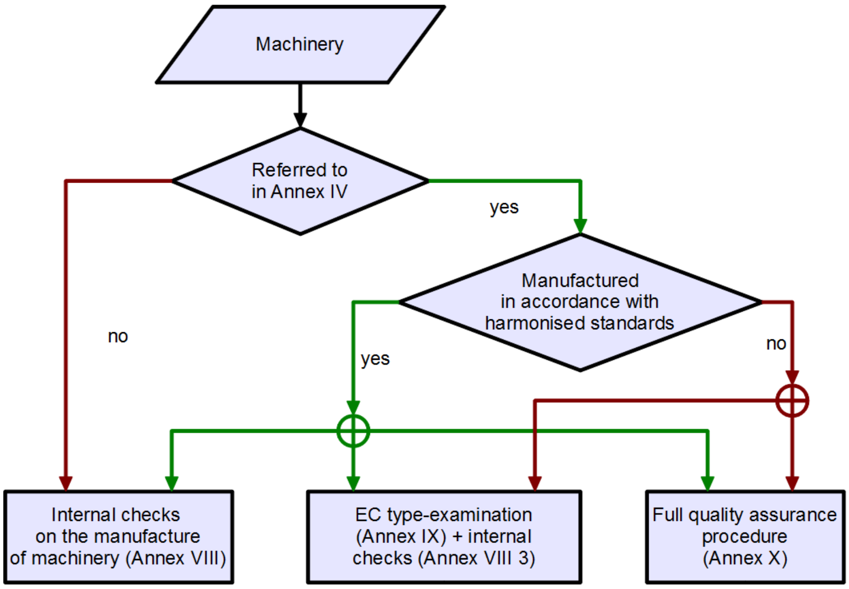

Machinery
Referred to in Annex IV
yes
no
Manufactured in accordance with harmonised standards
yes
no
Internal checks on the manufacture of machinery (Annex VIII)
EC type-examination (Annex IX) + internal checks (Annex VIII 3)
Full quality assurance procedure (Annex X)

Figure 2.2: Flow diagram depicting applicable conformance procedures depending on machine type and manufacturing process

tion principle[4]: whenever possible, the manufacturer has to eliminate risks by choosing a design that does not create the risk in the first place. Only when this is not possible may the risk-involving design be chosen, and appropriate protective measures be taken to reduce the risk.

Apart from such general principles, the annex contains concrete requirements that apply to all kinds of machinery, as well as rather specific requirements for machinery such as lifting machinery or machinery intended for underground work. Requirement 1.1.4 gives a good example of how specific the requirements get for well-established and well-understood problem areas. It is concerned with the lighting of machinery: *"Machinery must be supplied with integral lighting suitable for the operations concerned where the absence thereof is likely to cause a risk despite ambient lighting of normal intensity."* It then specifies characteristics of a proper lighting: *"Machinery must be designed and constructed so that there is no area of shadow likely to cause nuisance[...]"*. Several other requirements relating, among others, to hazards due to fire, explosion, noise, machinery movement, or even lightning are stipulated.

However, no requirements are given for safety components, let alone logic devices. To show compliance with the directive for these, one has to refer to the relevant standards. Revisiting the setting conceived at the beginning of Sec. 2.1, it can be said that no concrete requirements on AGVs can be found either. However, paragraph 3.3.3, concerned with self-propelled machinery, contains the relevant sentence from which all else follows: *"Remote-controlled machinery must be equipped with devices for stopping operation automatically and immediately and for preventing potentially dangerous operation[...]"*. Again, to

[4] also called *inherently safe machinery design and construction* by the directive

show compliance the relevant standards need to be resorted to.

The remaining annexes II to XII have been alluded to above or are not relevant in the context of this thesis.

2.1.2 The Safety Standard IEC 61508

IEC 61508 – Functional safety of electric/electronic/programmable electronic safety-related systems[80] is an international standard published by the International Electrotechnical Commission (IEC). It is concerned with all safety-related aspects that arise when electric, electronic or programmable electronic (E/E/PE) systems are used to perform safety functions. This standard is a basic safety publication, meaning that its content must be taken into account during the conception of other IEC safety standards. It is both intended to be used as a template for more specific sector or product standards, and to be applied on its own in sectors where there is no specific standard available. Major standards derived from IEC 61508 are, e. g., ISO 26262, the upcoming safety standard for the automotive sector, the nuclear sector safety standard IEC 61513, or the safety standard for E/E/PE control systems EN 62061, which is a harmonised standard according to the machinery directive. IEC 61508 is a *process oriented* standard in that it considers all safety-related aspects of the whole life cycle of an E/E/PE system, from the conception phase through design, implementation and commissioning to the maintenance and dismantling phases. This distinguishes it from less modern standards that are mainly product oriented and focus on the assessment of failure probabilities of the developed system designs. The scope is *functional safety*, which is that part of overall system safety that depends on the correct and reliable functioning of E/E/PE parts of the system under development or of additional safety components. The ever-growing importance of the use of software in safety-related systems is acknowledged by the standard and *software safety requirements* are laid down in detail, in IEC 61508-3.

Important Terms & Concepts

We now clarify the definitions of some important terms as used in the standard and as generally applicable in the context of safety-related system development. The following paragraphs highlight the most important concepts introduced or assumed therein. Since software verification is our major concern, we then directly move on to the covered software aspects, eliding details concerned with hardware.

1. Underlying every other concept is that of a *harm*. Harm needs to be avoided, and this is what safety is all about. Harm is defined as the injury or health impairment of persons. It may be caused directly or via the damaging of goods or the environment.

2. A *hazard* is a potential source of harm. Besides taking measures to eliminate or bound hazards, a vital part of the safety life cycle is the identification of hazards.

3. Next comes the concept of a *risk*; it is defined as the combination of the probability and the potential magnitude of a harm. The standard does

not discuss what "combination" precisely means, but assumes in examples that the magnitude of a harm can be quantified and then combines the two dimensions via multiplication. In any case, it is important to keep in mind when talking about risks that two dimensions are always involved: probability and severity.

4. *Safety* describes the absence of unjustifiable risks. Yes, it can be phrased so simply.

5. A *safety function* is a function that is performed by an E/E/PE (or other) system to maintain or reach a safe state of the *equipment under control (EUC)* with a view to specified hazardous events. The canonical example of a safety measure that does not constitute a safety function is safety-by-construction, i. e. an inherently safe design avoiding particular hazards altogether. On the other hand, a typical safety function would be the switching off of a heating unit whenever the measured pressure in a kettle exceeds a specified maximum.

6. *Safety integrity* measures the probability that a safety-related system is able to perform the safety functions assigned to it.

7. Finally, *verification* is defined as the process of ensuring by examination and documentation that the requirements have been met. It is to be distinguished from *validation*, which is the process of ensuring by the same means that the specific requirements for an intended use are satisfied. The provided distinction is rather subtle, but is similar in spirit to the well-known dichotomy into the two questions "Did we build the system right?" (verification) and "Did we build the right system?" (validation). In remarks it is made clear what the most important distinction is: verification applies to all development phases and ensures that the concrete requirements for a particular phase are adhered to as well as that the specified outputs are produced correctly. During the validation process one checks that the top-level safety requirements are satisfied by the final system in a specified application scenario. We will further distinguish between verification and *formal verification* in this thesis. By the latter we denote that part of the overall verification in which formal methods are applied.

Safety integrity levels One of the central concepts introduced by the standard is that of the safety integrity level (*SIL*). Four levels of safety integrity are defined (SIL 1 to SIL 4). They represent a discretised view upon the requirements on the safety integrity of systems executing safety functions. The standard uses these safety integrity levels for classifying requirements: a requirement is either generally applicable to all safety-related systems, or targeted at those having a particular SIL assigned to them. The SIL assigned to a safety-related system —called the *target* SIL— therefore essentially dictates which safety requirements it must satisfy, and how demanding they will be.

An SIL is not an inherent property of a system, but of a safety function, which is executed by one or more specific safety-related systems. It can be understood as an evaluation of the importance of the correct functioning of those safety-related systems for the overall safety[140]. Target SILs are determined in the course of a risk assessment. It can be said that the standard actually

SIL	Low demand mode	High demand mode
	(Prob. of failure on demand)	(hazardous failures / hr)
4	$10^{-5} \leq P_{fd} < 10^{-4}$	$10^{-9} \leq F_h < 10^{-8}$
3	$10^{-4} \leq P_{fd} < 10^{-3}$	$10^{-8} \leq F_h < 10^{-7}$
2	$10^{-3} \leq P_{fd} < 10^{-2}$	$10^{-7} \leq F_h < 10^{-6}$
1	$10^{-2} \leq P_{fd} < 10^{-1}$	$10^{-6} \leq F_h < 10^{-5}$

Figure 2.3: Assigning an SIL to safety functions in low or high demand modes

pays more attention to safety integrity than to the safety functions themselves. This is partly due to the fact that we are dealing with a generic standard, which cannot be concerned with measures against specific hazards. But moreover, the viewpoint is that it is easier to perform a hazard analysis and uncover the necessary safety functions, than to "get these safety functions right", that is, to establish the necessary safety integrity for them.

Now the question is: how can a target SIL be established for a given safety-related system? The target SIL crucially depends on an evaluation of the *maximum tolerable risk* that one is willing (and legally allowed) to accept for a given system — where system means the EUC, from which the risk originates, and not a particular subsystem or external safety component executing safety functions. Whether a risk is tolerable or not can only be decided taking technological (How much safety can currently reasonably be achieved? Are there other technologies achieving higher safety and similar functionality?), social (How are benefits and risks of a technology perceived in society?), as well as political (Is the development of the technology considered important?) and legal (Are national and international regulations met?) considerations into account. Comparisons with other tolerated risks are also in order. For example, [146] estimates the annual risk of fatalities to employees in the UK as 1 in 125000. While in the agriculture, hunting, forestry and fishing sector the risk even lies at 1 in 17200, the service sector tends to be a much safer workplace, with an annual fatality risk of only 1 in 333000. Different sectors will therefore accept different risks based on past experiences and present situations.

Sometimes, the tolerable risk of an EUC can be expressed in terms of a maximum fatality (or injury) rate per annum (p. a.). For the cases in which a such quantification is possible, IEC 61508 provides a table from which the corresponding SIL can be read off; it is reproduced here as Fig. 2.3. When assigning an SIL there are two different *operation modes* of the safety function to consider: low demand mode, in which the safety function will only be requested less than once a year (e. g., a car airbag), and high or continuous demand mode, in which it is executed more often or permanently (e. g., collision avoidance for AGVs). To illuminate the use of the table, assume that for an AGV the maximum rate of hazardous events (person injured by AGV) is $1.5 \cdot 10^{-5}$, and assume further that an analysis based on data about a previous model demonstrates that only $4 \cdot 10^{-2}$ of the hazardous situations (person crossing driveway) lead to a hazardous event. This yields a maximum tolerable failure rate of $1.5 \cdot 10^{-5}/(4 \cdot 10^{-2}) = \frac{3}{8} \cdot 10^{-3}$ p. a., or approximately $4.3 \cdot 10^{-8}$ hazardous events per hour. Hence, the target is SIL 3 in this case.

Safety life cycle Another important aspect of IEC 61508 is that it is not only concerned with technical properties that concrete products must satisfy, like levels of hardware redundancy or the use of a particular programming language, but that it explicitly acknowledges that safety can only be achieved if it is taken into account during the whole system life cycle. The standard is therefore process oriented rather than product oriented. It proposes a general safety life cycle, where for each phase several requirements are specified. Importantly, this life cycle refers to the development of the overall system (comprising the EUC and all its safety components) and not only to the components performing safety functions. It also proposes an E/E/PE system safety life cycle, a software safety life cycle and a software development model; neither these nor the general safety life cycle have prescriptive character so that other models can be used. However, it then needs to shown that all requirements set out for the reference model are also considered by the alternative. The following safety life cycle phases are described in IEC 61508-1. The software development model is covered further below.

1. Concept phase — the main goal here is to gain an understanding of the EUC and its environment.

2. Definition of the application area — this is in particular necessary to fix the limits of the system and to deliver these as inputs to the hazard analysis phase.

3. Hazard analysis and risk assessment — through these, the necessary safety functions and their safety integrity are determined. They may also affect the system design by discovering that further risk elimination is necessary via design changes.

4. Safety requirements specification — produces the essential safety requirements specification (*SRS*) documents, from which all further safety requirements must be derived.

5. Assignment of safety requirements to specific (E/E/PE or other) systems — during this phase the safety functions are turned into concrete systems realising them.

6. Planning: operation, safety validation, commissioning — the major planning effort is devoted to the specification of test procedures for the validation of all safety functions and their safety integrity.

7. Realisation of safety-related systems — the realisation of other systems is irrelevant for the safety life cycle.

8. Commissioning

9. Safety validation — ensures that the safety requirements set up in the SRS are satisfied. It is performed according to the safety validation plans.

10. Operation, maintenance and modification

11. Decommissioning

Quantitative and qualitative approaches to safety It is well known that for most hardware components *random failures* are the predominant kind of failure, where causes are of a physical nature, e. g. material fatigue or alpha radiation. Software, on the other hand only suffers from *systematic failures*: all causes of software failures are built into the software from the start and are hence due to design or implementation flaws[5]. In the presence of such systematic failures a quantitative approach to achieving safety is not appropriate (though not completely out of the question, as argued in [69]), and qualitative measures must be considered as well. These include the use of safety life cycles —as described above— whose activities are believed to reduce the number of systematic failures; the prescription of concrete measures that need to be taken during realisation, e. g. particular code coverage measures or the setting up of a configuration management; and even requirements on the project management, like the assignment of responsibilities or, more generally, the implementation of a quality assurance system. IEC 61508 acknowledges this fact and —with the single exception of assessing "proven in use" software— takes a purely qualitative approach to software safety. The underlying assumption is that by adhering to the qualitative requirements of a particular SIL, the failure rate of the software can be brought down to the corresponding hardware failure rate of that SIL.

This dichotomy is also present in SIL targeting: above, we described a quantitative approach to assigning a SIL to a safety function. However, the standard also allows a qualitative approach involving the use of risk graphs, in which the relevant categories of severity of possible harm, frequency of exposure of persons to hazards, and alternatives to avoid the danger are discretely classified (e. g. "avoidance possible by leaping aside" and "avoidance impossible") and then a matrix or graph yields SILs for each of the possible combinations.

IEC 61508-3: Requirements on Software

Having pointed out that IEC 61508 takes a qualitative approach to software safety, we now want to further investigate the software safety life cycle according to which software development has to be done. As before, deviations are possible, but require explanation and justification.

V-model The development process described in IEC 61508-3 is a variation of the well-known V-model[32], a process model essentially based on the waterfall model. It is depicted as defined in the standard in Fig. 2.4. It obtains its name because it separates the life cycle into two major blocks (forming the legs of the "V"): an initial block of activities concerned with the gathering of high-level requirements and their decomposition, or refinement, into a software design and finally a concrete program; and a subsequent block in which the program is integrated into the overall system and verification at all necessary levels of abstraction ensures that the requirements set up in the first leg are satisfied. The final validation of the integrated software in its intended application area eventually ensures that all high-level safety requirements are met. Obviously,

[5] Of course, with modern microprocessors the distinction between 'simple' hardware and 'complex' software no longer holds (if ever it did); however, the functionality of the hardware used in safety-related systems is usually well-understood and rather comprehensively tested. It also has to be shamefully admitted that hardware designs are commonly put under more rigorous scrutiny than software designs[131].

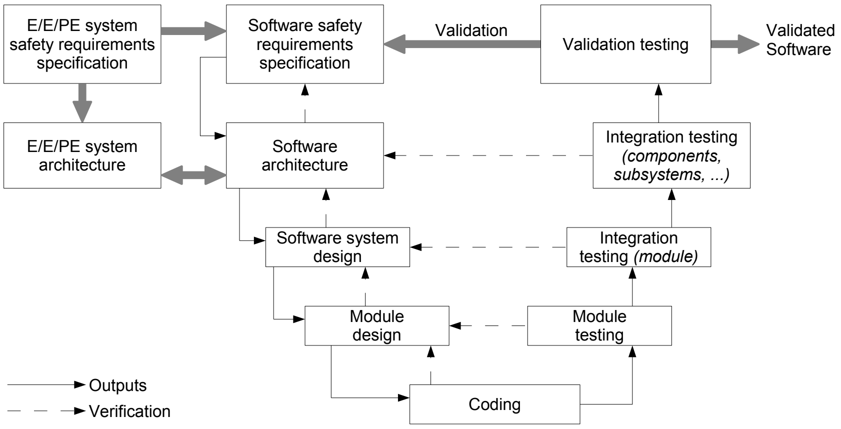

E/E/PE system safety requirements specification
Software safety requirements specification
Validation
Validation testing
Validated Software
E/E/PE system architecture
Software architecture
Integration testing *(components, subsystems, ...)*
Software system design
Integration testing *(module)*
Module design
Module testing
Coding
Outputs
Verification

Figure 2.4: Software development process as of IEC 61508-3: the *V-model*

its sequential character does not inhibit the need for iteration whenever violated or unsatisfied requirements are discovered.

General requirements Structurally, Part 3 also proceeds by attaching requirements to each phase of the software life cycle. Compliance with these provisions is practically demonstrated by creating tables for each set of requirements, i. e. for each life cycle phase, and filling out these tables with arguments or links to arguments about how the particular requirement has been met. For specific measures that need to be taken during design, development, verification and validation a set of tables is already given in annexes. The highlights of these are discussed below.

No striking novelties or peculiarities can be found among the more general requirements. These rather sum up standard software engineering practices. Examples are the requirement of traceability from software safety requirements back to the SRS, or from lower-level design requirements back to ones defined for the software architecture. Traceability is a big issue in safety-related systems, and one of the better selling points for formal methods. Traceability implies two desired properties of a system: it is a means to make sure that the concrete design or product satisfies the high-level requirements, a property one might call *correctness* of the system; and it wards off the integration of additional functionality into the system, since every feature has to be justified by at least one high-level requirement. The latter property might be called enforced *limitation* of the system.

Other requirements include the need to regard the whole software running on a safety-related system as safety-related, unless sufficient independence between software modules can be proven, or the claim for a software configuration management.

Measures Several tables in Annex A and B provide concrete normative guidance on what measures to apply in different phases of the software development life cycle. We list here the ones that are most relevant for consideration in the design of a formal verification environment. Measures related to testing, of which there are quite a few, are not included, as they are separately discussed in the following paragraph.

- Error detection needs to be taken into account in the software architecture, where the term "error" refers to those within the software itself, i. e. possible causes of systematic failures, and not to external (hardware) failures. Measures considered appropriate are the use of error detecting codes[6], failure assertion programming, and the dynamic supervision of control and data flow. We notice that these measures can certainly be subsumed by the use of formal verification at the code level, in the sense that all such errors for which a programmable detection mechanism and a workaround exists can be avoided in the first place.

- A strongly typed programming language is supposed to be used at higher SILs. Five high-level programming languages are explicitly mentioned. They are Ada, Modula-2, Pascal, Fortran 77, and C.[7] For all languages the restriction to language subsets is advocated, where for C it is unconditionally required to furthermore use coding guidelines and static analysis tools. C apparently made it into the standard despite its under-definedness and its many pitfalls because of its heavy use in industry and the tremendous tool support available[73].

- The use of features considered to impede verification is discouraged: recursion, dynamic memory, the unrestricted use of pointers, and unconditional jumps should be avoided.

- Certified tools, particularly compilers, must be used. The certification of these tools has to be done according to recognised standards, which are not defined any further.

- The final safety assessment shall be guided by the results of a failure mode, effects and criticality analysis (*FMECA*) and of a fault tree analysis (*FTA*), which are therefore necessary.

Use of formal methods Several concrete formal methods are listed and explained in Part 7 of the standard. They are: CCS[107], CSP[77], HOL[66], LOTOS[27], OBJ[65], temporal logic[129], VDM[86] and Z[141]. Curiously, these are all rather heavy-weight modelling or verification techniques. Industry standard techniques like model checking and automated code generation from formal or semi-formal system models are not appreciated by IEC 61508, which sets it in sharp contrast to the upcoming revision of DO-178C *Software considerations in airborne systems and equipment certification*—the standard

[6] This appears a bit irritating at first, since such error detecting codes are more common and appropriate for the detection of hardware failures. However, one can imagine using them to ensure data integrity across calls to "untrusted" functions, etc.

[7] The use of ladder logic and other primitive languages is also promoted, but this obviously only applies to software of lower complexity.

for software certification in avionics— in which the adoption of these modern development and verification techniques is one explicit focus[104].

From the viewpoint of a formal methods proponent, one can say that formal methods are fighting on two fronts for industrial recognition and appreciation by standards like IEC 61508: On the one hand, they can be used in the design and early development phases, i. e. on the downward leg of the V model. Here, in virtually all cases where formal methods are called for, there is the alternative of applying *semi-formal methods* (block/flow diagrams, state charts, truth tables, Petri nets, etc.), where semi-formal methods are highly recommended from SIL 3 upwards, while for formal methods this is the case only for SIL 4. Even *structured (development) methods* (Jackson System Development or Yourdon, among others), that establish no technical properties of the system whatsoever and are primarily concerned with the development process itself, are a highly recommended alternative for all safety integrity levels.

On the other hand, formal methods can be used for verification and validation, i. e. on the upward leg. Here, the traditional alternative is verification by testing. It is interesting to note that while the standard mentions formal methods quite prominently with respect to the design phases, testing definitely plays the primary role in the verification phases[8]. However, in a remark about requirements on module design[9] it is noted that the amount of functional testing necessary can be reduced through the use of formal methods. The reduction is not quantified. There are several required measures involving testing for the subsequent phases of module integration and hardware/software integration for which no formal equivalent exists. Examples include the requests for dynamic analysis, black-box testing, performance testing, or statistical testing.

During safety validation testing again must be the primary applied measure[10], but may be supported by simulation and modelling, where the latter also allows formal modelling techniques.

Even though the standard explicitly distinguishes static analysis and formal methods, it is obvious that formal software verification subsumes several static analysis techniques. The latter are highly recommended for software verification from SIL 2 upwards. We elaborate on the issue of covering required measures through the application of the verification environment presented in this thesis further in Sec. 7.5.

2.2 The Theorem Prover Isabelle

Isabelle[11] is a generic proof assistant. It enables the machine-assisted formalisation of logical calculi, called *object logics* such as first-order logic, set theory, or higher-order logic. At the foundation of this framework lies a minimalist higher-order *meta-logic* [123] that provides higher-order unification and resolution as the basic inference mechanisms as well as implication (written $\Longrightarrow$), universal

[8] This becomes manifest in a remark (§ 7.9.2.12) on verification, where it is said that static analyses in general are only considered appropriate in the early life cycle phases, while after having assured the correctness of all individual software modules, the test is the primary means of verification. This statement is repeated in a comment on Tab. A.9.

[9] § 7.4.7.2, Remark 2

[10] This is literally requested in § 7.7.2.6.

[11] Isabelle has been publicly available since 1994. Updates have been published annually in recent years. We developed the verification environment using version Isabelle-2009.

quantification ($\bigwedge$) and equality ($\equiv$) as the basic connectives. These are used to encode natural deduction style inference rules of the respective object logic. For example, the usual introduction rule for the universal quantifier in higher-order logic

$$\dfrac{\begin{array}{c}[x]\\ \vdots\\ P\ x\end{array}}{\forall x.\ P\ x}$$

is formalised in Isabelle's meta-logic as

$$(\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x \tag{2.1}$$

The meta-level quantifier expresses the side-condition that the parameter x used in the derivation of $P\ x$ must be fresh, i. e., not appear in any contextual assumption. Meta-implication, on the other hand, encodes inference steps in the object logic, while meta-equality is primarily used to encode axiomatic definitions. Rules involving several assumptions, as in the conjunction introduction rule, are encoded by chains of implications, but are usually written in an abbreviated form using brackets. The following two notations for the conjunction introduction rule (named *conjI*) are therefore equivalent:

$$\begin{gathered} A \Longrightarrow B \Longrightarrow A \wedge B \\ [\![\ A;\ B\ ]\!] \Longrightarrow A \wedge B \end{gathered} \tag{2.2}$$

Isabelle provides comprehensive support for inference in excess of simple rule application in the form of a powerful conditional rewriting engine called the *simplifier*, a classical tableau prover called *blast*, and decision procedures for linear integer arithmetic, among others. Moreover, it is possible to extend the reasoning capabilities programmatically in nearly arbitrary ways by defining so-called *tactics*, which are functions mapping theorems to theorems, written in Isabelle's implementation language ML. The soundness of such extensions is guaranteed by the well-known *LCF system approach* [67] which employs the data abstraction facilities of ML to restrict the access to the crucial datatypes such that only sound transformations are possible.

Isabelle also provides a rich machinery for rewriting-based syntax tree transformations and simpler mixfix annotations that allows us to keep definitions and specifications close to mathematical conventions and thereby readable. During development, the associated ProofGeneral module for the Emacs editor allows an appropriate display of theories using a large glyph set. An integrated document preparation system allows to transform theory definitions into PDF documents. We exploit this in several places of this thesis by inserting parts of Isabelle theories verbatim, instead of rephrasing the respective definition in a more conventional style.

Our work is based on Isabelle/HOL, a formalisation of higher-order logic which represents the most widespread object logic of Isabelle. The logic comes with a wealth of theory libraries, ranging from complex numbers over calculus to lattices, set theory and algebraic concepts like rings and fields. We will only cover the most relevant aspects of this logic here and refer to the excellent tutorial [116] for further details.

2.2.1 Concepts

Isabelle/HOL is based on the simply typed lambda calculus and hence the notation slightly differs from that conventionally used in mathematical text books. It is instead rather close to what is used in functional programming languages like ML or Haskell.

Types

Types are built from the constructors ⇒ for function types, × for product types, and from predefined and user defined types. The latter can be basic types such as *bool* or those for the usual number types (*nat* for the natural numbers $\mathbb{N}$, *int* for the integers $\mathbb{Z}$, *real* for the real numbers $\mathbb{R}$, and *complex* for the complex numbers $\mathbb{C}$), but also include further type constructors such as that for sets over a given type α (written in postfix notation as α *set*), lists (α *list*), or those introduced by type definitions. Terms can be explicitly annotated with a type, as in $1 :: \mathit{nat}$, either as an explicit restriction or to serve as documentation. Types can be defined in several ways. We briefly describe the relevant ones used in this thesis.

Datatype definitions introduce inductive datatypes (or free algebraic datatypes) defined by a collection of constructors. Datatypes can be parametric, i. e., defined generically over arbitrary other types. The classical example is the datatype of lists:

```
datatype 'a list =
    Nil
  | Cons 'a "'a list"
```

Primed variables are used for type parameters; *'a* can be instantiated with any type, to form lists of integers, strings, etc. Syntax transformations allow us to write the empty list *Nil* as [] and a non-empty list such as *Cons* 1 (*Cons* 2 *Nil*) as 1#2#*Nil* or even shorter as $[1, 2]$. Injectivity and disjointness properties of the constructors are proven automatically by Isabelle.

It is possible to perform a case distinction on the constructor by which a value of the type was built via conventional pattern matching. For example, the test for emptiness of a list is defined thus:

```
definition null :: "'a list ⇒ bool"
where "null xs = (case xs of
                    [] ⇒ True
                  | (x#z) ⇒ False)"
```

Records can be viewed as tuples with named components for our purposes. The following definition introduces a record type for two-dimensional points:

```
record point =
   X :: real
   Y :: real
```

In addition to defining *point* as a new type, two accessor functions X and Y are introduced together with appropriate access and update theorems. Record literals are written $\llparenthesis X = a,\ Y = b \rrparenthesis$, accessing the X component of a record r is done by applying the accessor, $X\ r$, and functional updates on records are written $r\llparenthesis X := a\rrparenthesis$, yielding a record value that equals r on the Y component and has a as its X component.

Type definitions are the most elaborate form of introducing a type. They allow us to define a type whose values are isomorphic to a given set of elements of an existing type. For example, we could define a type of functions over natural numbers that yield values $\neq 0$ only on a finite domain:

```
typedef fin_seq =
  "{f :: nat ⇒ nat. ∃n. (∀m > n. f m = 0)}"
  by auto
```

The subsequent proof "by auto" verifies that our definition does not introduce an empty type, which would lead to an inconsistency.

Terms

Predicates, functions, applications and constants are all *terms* in Isabelle/HOL. The syntactically most obvious deviation from conventional mathematics is that function application is written simply by juxtaposition of the function term and its arguments, as already seen in Eq. (2.1), where $P\ x$ stands for the application of predicate P to its argument x. A predicate over a type α is simply a function of type $\alpha \Rightarrow \mathit{bool}$. Non-recursive functions are formed by lambda abstraction. For example, the function to add two integers is denoted by the term

$$(\lambda x\ y.\ x + y) :: \mathit{int} \Rightarrow \mathit{int} \Rightarrow \mathit{int}.$$

As usual in higher-order logic, most functions are defined in a *curried* form allowing for their partial application. For example, if the above function is bound to f, then $(f\ 1) :: \mathit{int} \Rightarrow \mathit{int}$ yields a function that will add 1 to its argument. Tuples can be, but seldom are, used to achieve the more conventional type $\mathit{int} \times \mathit{int} \Rightarrow \mathit{int}$ for binary functions, as in $\lambda(x, y).\ x + y$. Terms are named via definitions, as seen above for the function *null*.

Recursive function definitions Isabelle's function package allows us to define primitive recursive functions in a convenient way, as we would in a functional programming language. All necessary well-foundedness proofs are performed automatically and are invisible to the user. It is even possible, though not needed in this thesis, to define arbitrary recursive functions, if the user can supply a well-founded relation over the terms to which the function gets recursively applied. To provide yet another classical example, this is how the function computing the length of a list is defined:

```
fun length :: "'a list ⇒ nat"
where
"length [] = 0" |
"length (x#z) = 1 + length z"
```

Sets Isabelle identifies sets and predicates: both sets and predicates over values of type α have type $\alpha \Rightarrow \mathit{bool}$ and we can consider a predicate P as the set of all values satisfying P, i. e., $P = \{x.\ P\ x\}$. Nonetheless, Isabelle provides syntax for set notation and defines the standard set theoretic operations such as membership $x \in X$ (identified with $P\ x$), intersection $X \cap Y$, union $X \cup Y$, set difference $X - Y$, subset relationship $X \subseteq Y$, etc. The universal set is called *UNIV*, and the union over an index set is written as $\bigcup_{i \in I}(f\ i)$, making use of lambda as the ultimate binder: the term syntax-expands to $\mathit{UNION}\ I\ (\lambda i.\ f\ i)$, where *UNION* is a higher-order function satisfying the equation

$$\mathit{UNION}\ I\ f' = \{y.\ \exists x \in I.\ y \in f'\ x\}.$$

Partiality Isabelle/HOL is a logic of total functions, i. e., there is no such thing as an undefined term. Partiality can be modelled with the help of the option datatype, which extends a given type by an element *None*. It is defined as a regular datatype:

```
datatype 'a option =  None | Some 'a
```

For a 'partial' function $f :: \alpha \Rightarrow \beta\ \mathit{option}$ Isabelle allows us to write its type as $\alpha \rightharpoonup \beta$. Moreover, *dom f* denotes *f*'s domain, i. e., those values for which it does not yield *None*.

Another way to model partiality is by under-specification; the function package allows us to omit certain cases that are necessary for a complete function definition. An example is the function *the* satisfying *the* (*Some x*) = *x*, but whose value when applied to *None* remains unknown. (Even though it still denotes *some* value, since every function is semantically total.)

Theories, Definitions and Theorems

A formal development in Isabelle is structured into *theories*. A theory essentially consists of (type and term) definitions constituting the signature of the theory, as well as theorems. (The latter can also be named lemmas or corollaries; this distinction only applies at the syntactic level to be able to indicate their respective importance.) Theorems must be proven in one of two proof styles. By using the *structured* Isar proof style by Wenzel [152] it is possible to write human-readable proofs which make intermediate proof states explicit and which "verbalise" proof steps through the use of appropriate keywords. Fig. 2.5 gives an example Isar proof of the fact that the length of two concatenated lists is the sum of their individual lengths.[12] The keyword **lemma** (as well as **theorem** and **corollary**) introduces a new theorem named *length-append* and requires a subsequent proof of its statement. The proof is by induction over *xs* (**proof** (*induct xs*)), which introduces two cases. The case for the empty list is shown first. It can be immediately proven by Isabelle's simplifier (**by** *simp*). The second case requires us to prove the statement *for all* lists *a # xs*, where *a* is an arbitrary list element and where the induction hypothesis holds for *xs*. The **fix** keyword introduces fresh variables for the list head *a* and the tail *xs*; this is analogous to the step in a textbook proof where one would say *"let a and xs be arbitrary fresh entities"*. The **assume**d hypothesis is explicitly written down subsequently, which already allows us to conclude the second case, again by a simple call to the simplifier. The proof is ended by the **qed** keyword.

The other proof style is called the *tactic style*, which is of a more imperative nature. Here, unfinished proofs are in a current *state* consisting of one or more *proof goals* which can be modified by applying tactics that yield a new state

[12] Where the concatenation of lists *xs* and *ys* is written infix as *xs* @ *ys*.

```
lemma length_append:
  "length (xs @ ys) = length xs + length ys"
proof (induct xs)
  show "length ([] @ ys) = length [] + length ys"
    by simp
next
  fix a xs
  assume A1: "length (xs @ ys) = length xs + length ys"
  thus "length ((a # xs) @ ys) = length (a # xs) + length ys"
    by simp
qed
```

Figure 2.5: An example proof in the structured Isar proof language

```
lemma length_append':
  "length (xs @ ys) = length xs + length ys"
  apply (induct xs)
  apply simp
  apply simp
done
```

Figure 2.6: The same theorem proven in the tactic style

and new (hopefully simpler) goals. For example, if the current proof goal is a conjunction $A \wedge B$ and rule *conjI* of Eq. (2.2) is applied, the subsequent state comprises the two goals A and B, i. e., the hypotheses of the applied rule. Proofs are hence performed backwards. The corresponding proof of *length-append* in the tactic style is given in Fig. 2.6. The obvious difference is that the intermediate proof states are not contained in the proof script. Rather, the **apply** command is used to, well, apply specific proof procedures (or *tactics*) that modify or discharge proof goals in the current implicit proof state. Intermediate proof states can only be inspected when the proof is manually replayed in the theorem prover.

Depending on the kind of proof, this is either an advantage or a disadvantage. Mathematically pleasing proofs of properties of general interest certainly deserve a structured proof document that can be understood without requiring tool assistance to walk through single imperative proof steps. The correctness of a concrete program function w. r. t. its specification, on the other hand, does mostly not belong into this category. Here one is interested in a boolean result; one is satisfied if the proof succeeds by any means – within the bounds of what the prover allows, of course. We therefore tailored the support for correctness proofs provided by the verification environment towards the tactic proof style. This particularly relieves the verifier from having to write down the many intermediate proof states. Interesting properties of the formalisation itself (e. g., the split heap property of the memory model), however, were often proved in the structured Isar style.

2.2.2 Presentation of Formal Proofs

We do not include Isabelle proofs in the written part of the thesis, as machine-checked proofs are often rather detailed and lengthy. Instead, we resort to a more digestible style in which only the most important proof steps are discussed. The complete formalisation including all proofs can be found in electronic form on the medium accompanying this thesis. We regard it as one of the true benefits of using a theorem prover that we are able to split the presentation of interesting properties from their detailed proofs. In pen-and-paper developments one would rightfully demand more proof details, to verify that the author did not miss allegedly easy cases or reasoned erroneously otherwise. Whenever a theorem is typeset as an Isabelle theorem in this thesis, like *append-length* above, it has been formally proven and its proof is part of the verification environment, if not stated otherwise.

Likewise, all definitions that are presented in Isabelle typeface, such as *null* and *length* in this section, are actually part of the verification environment.

Chapter 3

Language for Functional Specification

In this chapter the specification language that we developed to express functional properties of C programs is described and the approach taken is delineated against other relevant approaches to formal program specification. The most distinguishing features of the specification language are presented in detail: specifications as pre- and postconditions of program functions; a term language for specifying admissible memory layouts; the embedding of Isabelle expressions into specifications to achieve higher abstraction; and the modifications to C's type system that apply within specifications.

The formal semantics of specifications, as given by the formalisation in Isabelle, are deferred until Sec. 5.5, because the memory model and semantics of C programs have not been defined yet. This ordering was deliberately chosen, since the specification language must be comprehensible to and usable by persons that are not involved in the (formal) verification process and thus should not require knowledge about the formalisation.

3.1 Classification

The goal of every specification language used in program verification is to describe a number of properties that a program must satisfy in order to perform its intended function faithfully in any foreseen circumstance. We would call a program with the latter property "correct". Virtually no specification language is expressive enough to fully cover every aspect of program correctness; moreover, and more importantly, no concrete specification will ever be that complete and all-embracing. Hence, satisfaction of specifications is always a necessary criterion for program correctness, but hardly ever a sufficient one. Therefore every specification language needs to make its intended application area, hence its scope, very clear. We further need to distinguish between informal specifications, which are mainly expressed in natural language, augmented with drawings, diagrams and the like, and formal ones. A formal specification language provides a mathematically precise semantics for each language construct and each term of the language, and with the same precision defines what it means for a program to satisfy a given specification. We are only concerned with formal specification languages in this thesis and will henceforth omit the adjective “formal”, only using it when an explicit distinction against informal specifications is needed.

To clarify the scope of the developed specification language —to which we will refer as CSI (C Specifications with Isabelle/HOL) from now on— a view at other well-known approaches to program specification is in order. Hatcliff et al. [72], among others, categorise specification languages according to the level of system abstraction that the language assumes. At the top-level of system requirements the system is mostly viewed as a black box, or a collection of black boxes, with all implementation details omitted. Properties of such a system can be stated, among others, in terms of statecharts[70], or in the case of multiple communicating systems in terms of, e. g., CSP[77]. Mostly, however, requirements at this level are stated in natural language. Deeper properties can be stated and examined on what one may call the analysis level, in which models of the application domain and the system are built and analysed. On the architectural level, properties of software systems are predominantly specified in dialects of the UML and provide a static view of how the components of a larger software system are related. Formal verification is not common on this level. Finally, a specification language can directly relate to the concrete source code implementing a software system. CSI falls into this category. Like most other languages on this level, CSI focuses on the specification of functional properties of programs.

This leads to another axis on which specification languages can be categorised, namely the kind of behaviour that the language intends to specify. We encounter three major classes of behaviour amenable to specification:

1. Properties of the *functional* behaviour of a system are concerned with the system’s input-output behaviour as well as invariant properties of the system’s internal state during operation. Languages used on the analysis level include Alloy and Z[83, 141], which are both languages based on set theory, and the algebraic specification language CASL[23]. An analysis level specification abstracts away from implementation details like the concrete shape of data structures and operations thereupon, but rather models them with simpler mathematical objects like relations or functions. The essential operations of the system and their required properties as derivable from system requirements documents would, however, be formalised. It is common to subsequently refine such models into one or more detailed models, so that more and more specific design decisions are captured in the model. Ideally, one finally ends up with a model that reflects enough design decisions so that an implementation can directly be derived from it, possibly even in an automatic way. A recent industrially successful such methodology based on Z and model refinement is described by Barnes et al. [11]. The huge advantage of specifying at the analysis level is that one need not be bothered with technical details of the concrete implementation like memory management, algorithm efficiency or data layout and can instead concentrate on those properties relevant to the application domain. For safety-related systems, however, it is of particular importance to ensure that the actual software running in the system is correct. During the transition from model to code, whether it is done by hand or automatically by code generators, there is always the possibility of introducing

errors. Tools like the Astrée static analyser [25] are therefore specifically targeting the analysis of code automatically generated from models.

The most widely known variant of functional specification on the source code level goes by many names, like the pre-/post technique[72], design by contract[106], or behavioural interface specifications[94]. They are all based on the fundamental concept of specifying the properties of program operations —functions in procedural code and methods in object-oriented code— by two things: firstly, the requirements that need to be satisfied when an operation is executed (these are called the operation's *precondition(s)*), and secondly, it is stated what will be true after this operation has finished its execution when started in a state satisfying the preconditions (the operation's *postcondition(s)*). This approach to specification is based on the notion of axiomatic program semantics, as conceived in the seminal papers by Floyd [59] and Hoare [76]. Popular behavioural interface specification languages similar to CSI are JML[33] and ACSL[18]. Pre-/postconditions in these languages are formulated in the expression syntax of the programming language they apply to, where these expressions are usually extended with quantifiers, defined predicates and special symbols denoting various entities like, e. g., a function's return value. There is tool support for the verification of the functional properties thus postulated: the Spec# programming system[13], the SPARK tool suite[12], and the Why tool[55] are probably the most well known specimen. CSI and the verification environment described in this thesis are conceptually closely related to the latter two systems, although the focus on the kind of properties to be specified and verified differs.

2. When *temporal* properties are analysed, the system is most often viewed as some kind of labelled transition system, being in one out of a given (most often finite) set of possible states. The system performs transitions between states as a reaction to particular events. This model is particularly appropriate to describe reactive systems, which run indefinitely. An AGV controller might be described in this way, where states would include *Braking*, *Halted*, or *SilentRunning*, and a possible event might be *ObstacleDetected*. Languages like LTL or CTL can be used to specify desired temporal properties of the model, e. g. that after every *ObstacleDetected* event the system eventually reaches the *Halted* state. Model checking techniques allow to (automatically) ensure the model actually satisfies those properties [42]. In the case of LTL and CTL, the term "temporal" merely refers to the fact that sequences of events are considered that occur sequentially in time. If concrete real-time constraints need to be specified and verified (e. g., "operation P must follow Q after less than k ms"), languages based on timed automata or Time Petri Nets can be used[7, 21, 127]. Jhala and Majumdar [85] give an overview of software model checking techniques, i. e. verification of temporal properties of concrete source code.

   An important distinction between the functional and the temporal analysis of a system is that the former is more concerned with data, while the latter focuses on control. A functional specification takes a component of the system (e. g., a program function) and describes its black-box behaviour in terms of a (one-to-one, or general) relation between input and output data. Transition-based models, in contrast, provide an abstract view of the system's internals, in particular its discrete state, and which (abstract) events lead to state changes. Hybrid dynamic systems[6] allow the combined modelling of discrete and continuous behaviours by allowing to describe transition conditions and internal states of systems by (differential) equations. They are therefore well-suited for analysis-level specifications of systems like a vehicle controller, where there is a combination of discrete behaviour (the controller as a digital circuit) and continuous behaviour (the physical motion of the vehicle). The bias of hybrid systems modelling, however, is not the specification and verification of specific computations, as is the case for a language like CSI.

3. Sometimes it is important to specify the behaviour of a system with respect to certain *resources*, like memory, I/O channels, synchronisation locks, or time; for the latter two cases, timed automata, CSP and other process algebras are often used. The former two are predominantly analysed on the source code level, as they relate to implementation specific properties. A currently popular approach is the use of type systems that allow the static derivation of bounds for resource consumption[8]. Resource consumption properties cannot be specified with CSI.

3.2 Annotations

Syntactically as well as structurally, CSI strongly resembles JML and ACSL. Every function of the program or library is given a specification in terms of *annotations* that are embedded in the source code inside specially marked comments. This allows a tight coupling between specifications and source code, while allowing regular compilers to read the source files without the need for special preprocessors. As in the aforementioned languages, these comments begin with the character sequence `/*@` and end with `*/` or `@*/`. A function specification is placed immediately in front of the declaration or definition of the function it refers to. Function annotations are comprised of four annotation elements, which are also present in the example specification shown in Fig. 3.1:

1. The *precondition* of the function (often referred to as the `@requires` *clause*) is a predicate over the program state that is required to be true whenever the function is called. More specifically, a precondition is evaluated in the state in which the function's formal parameters have been allocated in memory and have been assigned the values of their actual parameters, but in which the local variables are not yet visible. This state is referred to as the *pre-state* of the function.

2. Requirements on the *memory layout* at function call time are further set forth in the `@memory` clause. Most importantly, it states the required sizes of arrays (which cannot be distinguished from single pointers by their types in C) and expresses separation constraints between memory areas, e. g., that two arrays may not overlap, or that a pointer may not point into a specific memory area. Semantically, such annotations simply add to the precondition of corresponding function.

```
/*@
  @requires (a == 0 || a == 1) &&
            (b == 0 || b == 1)
  @memory *out <*> *carry
  @modifies *out, *carry
  @ensures *out == (a + b) % 2
           *carry == a * b
  @*/
void halfadd(bit a, bit b,
             bit *out, bit *carry);
```

Figure 3.1: Functional specification of a half-adder: precondition, memory layout, modification frame and postcondition.

3. The *modification frame* (@modifies clause) describes the memory areas whose values might be changed by the execution of the function. Callers of the respective function can therefore be sure that all values outside the modification frame will keep their values across the execution of the function. The expressions denoting modification frames are evaluated in the same state as the precondition.

4. The @ensures clause specifies the *postcondition* of the function. It is evaluated in an (artificial) state in which the function's local variables are no longer visible, and where the formal parameters have re-obtained the values they had *before* execution of the function, i. e. in the state of the precondition. We call this state the *post-state* of the function. This predicate may reference the values of expressions in the pre-state via a special \old operator, as well as the function's return value via the keyword \ result .

Together, these annotations form the *contract* of the function:

1. It may only be called in program states and with arguments satisfying the precondition and the memory layout specification. If the function is called in a state violating this condition, none of the following guarantees are given;

2. its effects on memory are bounded by the modification frame,

3. the function will terminate, i. e. not get stuck in an infinite loop or recursion,

4. the function will be *safe* to execute, i. e. it will not perform invalid memory accesses or division by zero (see Sec. 4.2.2 for restrictions),

5. and finally, after execution of the function, the program will be in a state in which the postcondition holds.

While such an informal description of contracts will probably suffice for actually reading, understanding, and writing specifications in CSI, it is certainly insufficient for serving as a definition. We refer the reader to Sec. 6.2.1 for a complete formal definition.

From a verification-centred point of view, merely providing specifications for function interfaces is not sufficient. Two more constructs have been introduced that are used to annotate arbitrary, possibly compound, statements (the @join annotation) and loop statements (the @invariant annotation) in function bodies. Such annotations do not add to the interface specification of the functions containing them, but are solely used to enable the automatic computation of verification conditions (cf. Sec. 6.3). They are further described in Sec. 3.5, together with all other auxiliary specification items.

3.3 Specification Expressions

We collectively call terms occurring in pre-/postconditions as well as @join and @invariant annotations *specification expressions*, even though these annotations differ slightly in the set of allowed constructs, as we will see shortly. Specification expressions are essentially predicates over program states, where in the context of function specifications by a program state we solely mean the values of all objects existing in memory at a given time. This comprises all global objects (denoted by global variables), the parameters of all functions currently on the execution stack, as well as their local variables.[1] A program state in this sense does not include the contents of input or output channels like sockets or files. It also ignores the status of interrupts and other machine-related information that is not reflected in program variables. CSI specifications are only concerned with program memory and not with I/O.

The syntax of specification expressions is based on that of C expressions. This is done in virtually all approaches in the design by contract tradition, since it is very natural: we need access to the values of program variables in specifications anyway, and many properties are boolean combinations of arithmetic relations between these, as, e. g., in a.x <= max_val && 0 <= i. We do, however, not allow expressions whose evaluation causes the program state to change in specification expressions. Even though it is not our intention to check specifications at run-time as it is done in some design by contract methodologies — in which case side-effecting specifications cannot be treated transparently anymore, as their evaluation changes the program semantics —, we took this decision because side-effects rule out many extremely useful properties. For example, commutativity is lost: a++ == b && a > b is not the same as a > b && a++ == b: in a state where a == 1 and b == 1, the first expression would yield true (1 in C) and end up in a state where a == 2, while the second one would yield false (0) and not modify a's value.

The evaluation of C expressions may have nearly arbitrary side-effects, via the use of increment expressions (a++), function calls (f(a) − g()), or assignment (x = e) — which is treated as an expression syntactically. Therefore, it is not advisable to provide a definition of side-effect free specification expressions via a restriction of the (in any case informal) definition of C expressions. Instead, we give a constructive definition in terms of the abstract syntax of specification expressions.

[1] In scenarios where dynamic memory allocation is allowed, it would of course also include all allocated memory chunks.

3.3.1 Abstract Syntax

Specification expressions are built over a simple subset of C *lvalues*. The standard makes a distinction between lvalues as general object locators, and modifiable lvalues, which are lvalues that may appear on the left-hand side of an assignment[82, § 6.3.2.1]. Essentially, this excludes lvalues of array type, those with an incomplete type, and **const**-qualified lvalues. Under these general definitions, expressions like *f() or *(++x) can be modifiable lvalues. We prefer to keep object locators clean and simple, as given by the lval category of the abstract syntax shown in Fig. 3.2. Inside specifications, lvalues are built only from identifiers (ident), **struct** field selection, the dereferencing operator *, and array accesses. Such lvalues are the only entities which we may take the address of (category vexp). expr defines the category of specification expressions; they are formed over numeric literals (number), both of integral and floating type, and vexp. Additionally, the literal symbols \true and \false represent the values of the Boolean type _Bool. They further include unary and binary arithmetic operations, but no bit-shifting operations. The latter were excluded for simplicity, as they were not used in the algorithms we verified. Their inclusion would not pose any serious problems. For convenience, binary operators for implication --> and equivalence <-> were added. Instead of the conditional expression (e ? f : g) of C, we chose to use an arguably more readable conditional of the form **if** e **then** f **else** g. Casts are only allowed between what the standard calls *basic types* (cf. Fig. 3.4), and furthermore the newly introduced types _Bool as well as the mathematical types _Int and _Real (standing for mathematical unbounded integer and real numbers). We subsume all these types under the label *arithmetic types*. Sec. 3.3.3 defines casts (i. e., explicit) and implicit conversions in specification expressions.

The \forall and \exists quantifiers represent the first serious extension to C expressions. They both allow to quantify over the arithmetic types defined in the syntactic category btyp. Quantification over structured types or arrays is not allowed. A quantifier introduces one or more variables whose scope matches the quantified expression. Within, they are used like ordinary program variables, and all rules that apply to the latter also apply to the former, e. g., type conversion. In contrast to JML and ACSL, the expressivity of the language does not rest on the presence of these quantifiers, as explained in Sec. 3.3.2. They are mainly included for array-related specifications, where the quantified variables denote array indices, as in the classical specification of the maximum value of an array:

```
\exists unsigned int i; i < len && a[i] == max &&
  (\forall unsigned int j; j < len --> a[j] <= max)
```

Specifications whose domain of interest is more complex than pure arithmetic on scalars are specified with the help of Isabelle terms, which enables the quantification over values of arbitrary type, as we will see shortly.

Internally, i. e. in the Isabelle translation, every quantification over an arithmetic type is interpreted by a quantification over either mathematical integers (Isabelle type *int*) or mathematical real numbers (*real*) and an additional range restriction that directly corresponds the value range of the given type. For example, on a 32-bit machine, the above quantification would be translated to

```
"∃ (i :: int). 0 ≤ i ∧ i < 4294967295 ∧ ⋯"
```

```
lval ::= ident
       | lval.ident
       | *lval
       | lval[expr]

vexp ::= lval | & lval

binop ::= + | - | * | / | %
        | && | || | --> | <->
        | < | <= | == | != | >= | >

expr ::= number
       | \true | \false | \result
       | vexp
       | expr1 binop expr2
       | - expr | ! expr
       | \if expr1 \then expr2 \else expr3
       | (btyp) expr
       | \forall bindings; expr
       | \exists bindings; expr
       | \old expr
       | expr @ label
       | \valid lval
       | \array(lval, expr)
       | \separated(vexp1, expr1, vexp2, expr2)
       | \unrelated(vexp1, vexp2)
       | $ident
       | $ident(expr1, ..., exprN)
       | ^ident(expr1, ..., exprN)
       | ::ident(expr1, ..., exprN)
       | ${ isaterm1 ... isatermN }

bindings ::= btyp1 ident1, ..., btypN identN

btyp ::= _Int | _Bool | _Real
       | char | short | int | long | long long
       | unsigned char | unsigned short | ...
       | float | double | long double

isaterm ::= <raw isabelle text>
          | `{expr}
```

Figure 3.2: Abstract syntax of specification expressions

In postconditions and statement annotations it is often necessary to refer to values of expressions in states different from the ‘current’ one. A classical example illustrating the need is the specification of a function that increases a global variable gx by one: this is not a predicate over a single state, but relates one state to another. There are essentially two ways to deal with this problem: one can introduce auxiliary variables into the specification language, which are purely logical entities distinct from program variables. Their values are then typically specified in preconditions and can be referred to in postconditions. For an auxiliary variable N we might thus specify:

```
int gx;
/*@
  @requires gx < INT_MAX && gx == N
  @ensures gx == N + 1
  @*/
void inc_gx(void);
```

Even though auxiliary variables are a powerful concept that allows more than just the bookkeeping of old values of expressions [91], they perform badly in exactly this predominant usage pattern, because the back-reference to old values is done only in an indirect way. Further, they are not easy to reason about automatically, due to their existential nature. Therefore, we follow the JML tradition and use a more lightweight approach. The operator \old allows one to refer to the value of an expression in an appropriate previous state. In the case of postconditions, this is exactly the pre-state of the function. For @join annotations, it is the state before execution of the annotated statement. No other specification expressions may mention this operator. We specify the incrementing function thus:

```
int gx;
/*@
  @requires gx < INT_MAX
  @ensures gx == \old gx + 1
  @*/
void inc_gx(void);
```

Statement annotations sometimes also need to refer to values of expressions in arbitrary previous states. For example, in the invariant of a loop ‘pruning’ the values inside an array to some maximum value, one wants to state the invariant property that the values of all processed elements are either equal to the maximum or to their old value, whichever is smaller, while all unprocessed elements are unchanged. For these purposes, @join and @invariant annotations may use the @label operator, which evaluates its operand in the state which was active at the C label label:

```
loop:
  /*@
    @invariant ... && (\forall unsigned int j;
      (j > i && j < len --> a[j] == @loop(a[j])))
  while (i < len) {
    if (a[i] > MAX) { a[i] = MAX; }
    ++i ;
  }
```

Two further classes of language constructs remain: memory layout specifications and references to Isabelle/HOL terms. They are described in sections 3.4 and 3.3.2, respectively.

3.3.2 Embedding Isabelle/HOL

Our aim is to specify and verify program modules for the domain of safety-related robotics and automation. Functions in such programs often directly represent mathematical operations over the modelled domain. They range from simple vector operations (scalar product, transformations) over computing the convex hull of a point set to the approximation of the behaviour of a moving object. The set of data structures these operations work upon is rather restricted. For simplicity and memory safety, they tend to be static; dynamic objects are generally sparse, and disallowed in most safety contexts in any case. Predominantly, these data types are structurally just tuples and sequences of floating-point numbers and integers.

In contrast to their representations in programs, the objects of interest in the mathematical domain are not necessarily discrete and finite. They include time-dependent functions $f : \mathbb{R}^+ \to X$, arbitrarily shaped areas $A \subseteq \mathbb{R}^2$, convex polygons and so forth. We argue that specifications should be written in the language of this mathematical domain even at the low-lying module level we consider here, and not in terms of the 'program domain', i. e. as relations between values of program entities. A combination of more abstract, domain-related specifications and the strong guarantees w. r. t. adherence to these by the source code through the use of formal verification allows us to move the focus of discussions with reviewers from the concrete source code to the code/module specifications level. We have argued for this point in [61]. It is furthermore a well-known problem that specifications written in a language close or even equal to the programming language tend to be redundant, because they often simply reiterate what is in the code due to their lack of expressivity. As an example, it is not obvious how to specify a matrix inversion operation in terms of code entities, except by repeating how it was or would be programmed, so that one might end up with a specification like

```
    inv.a13 == -m.a11 * m.a13 - m.a21 * m.a23
 && inv.a21 == m.a12
 && inv.a23 == ...
```

which differs from its implementation only by the use of the equality operator == instead of the assignment operator =. Not much is gained by such a reiteration, and worse: it is a source of common cause failures, as the specification and the code look the same such that errors might either be detected in both, or be overlooked in both.

To obtain the desired degree of abstraction and detail in function specifications, an expressive, mathematically oriented language is required. Fundamental concepts like real numbers, sets, geometric transformations, but also concepts from analysis like derivations, integrals or limits should be easily definable or preferably predefined. Moreover, for the actual verification, a plethora of lemmas about these operations and their interaction will be needed. Also, for readability, the language should be syntactically flexible (e. g. support infix notation), and have a larger glyph set than plain ASCII: compare $\mathit{Int}(\mathit{Int}(X, Y), Z)$ to $X \cap Y \cap Z$, or even $\mathit{Subseteq}(\mathit{Union}(i, I, X[i]), \mathit{Img}(g, Z)))$ to $\bigcup_{i \in I} X_i \subseteq g\,`\,Z$.

As a final point, we do not expect to be able to prove the domain-related parts of program specifications automatically. The specifications we present in Sec. 7.3 are simply far beyond the capabilities of current automatic provers like the SMT solvers Z3 [51] or CVC3 [15], and even those of first-order reasoners like SPASS [151], despite the impressive advances these tools have made in recent years. This means that specifications will be visible not only to the specifier, but also to the verifier, who must understand them to successfully prove programs correct. Provers have their own input languages, which results in an encoding overhead: specifications must be translated to the prover's language. In principle, such encodings are incredibly hard to read when generated automatically.

These considerations led to the decision to directly let specifications contain Isabelle/HOL expressions. Isabelle/HOL satisfies several of the above criteria: it provides a flexible and powerful syntax machinery aimed at mathematical notation; large amounts of real analysis have been formalised, providing a good starting point for the further development of required theorems; finally, no discrepancy between specification expressions and their encoding in the prover's language is created if they coincide.

However, the aforementioned point about the need to refer to the program state in terms of the values of program expressions remains valid. Simple specifications like arithmetic relations between program variables, ranges of array indices, or validity of pointers are best written down in the C syntax. This resulted in a hybrid approach for CSI, where Isabelle and an extension of the C syntax can be combined by a quote/anti-quote-mechanism, letting us have the best of both worlds.

Embedding Single Functions

There are essentially two ways to embed Isabelle/HOL terms into specifications. Sometimes one only wants to refer to particular functions from the domain model that correspond to concepts used in the implementation. This can be done by referring to the function's name, prefixing it with a $. In the following example the Isabelle/HOL functions *min*, *max* :: *int* ⇒ *int* are referenced in this way:

```
/*@
  // ...
  @ensures *p == $min(a, b) && *q == $max(a, b)
  @*/
void twosort(int a, int b, int *p, int *q);
```

The arguments to $-functions are evaluated like the arguments in a C function call, i. e., in a call-by-value fashion. This type of embedding is suitable for simple functions that return C basic types and expect program values as arguments; the 'outer' syntax remains that of C expressions. A problem with this approach is that several program entities are not representable as values in C. The two most important ones are arrays and pointer-linked structures. For example, in the context of a declaration **struct** s { **int** *a; **int** *b; } s1; a call $foo(s1) would only make the *values* of s1.a and s1.b visible to *foo*, which are addresses. If the intention is to also pass the contents of these addresses to *foo*, a different embedding is necessary. The crucial question here is: how can parts of the program memory that are accessible by following pointers (or, in the case of arrays, by performing pointer arithmetic) be represented as Isabelle/HOL values? And further: how can we describe the way in which pointers are supposed to be followed to form such values? For example, we might want to include *s1.a in the value, but not *s1.b. A simple approach would be to provide one *read-memory* function for each C datatype. Our answer to this problem is simpler: we do not specify in advance how to build such representations, but simply pass the whole memory (program state) as an argument to Isabelle/HOL functions, which can then internally read the memory parts relevant to them. Such *state dependent* Isabelle/HOL functions are functions whose first argument is a program state. They are referred to in specifications by prefixing them with a ^. The expression ^bar(&s1) $<$ 1 refers to the state dependent function $\mathit{bar} :: \mathit{State} \Rightarrow \mathit{Loc} \Rightarrow \mathit{int}$, where *State* is the type of program states in the formalised memory model. The state that gets passed to *bar* is the same as the one in which the surrounding expression is evaluated.

(Anti-)Quotations

From the viewpoint of theoretical language expressivity, we could stop here, as it is now possible to define Isabelle/HOL predicates for the pre-/postcondition of every function, and simply refer to these in specifications, e. g., one could write function specifications like

```
/*@
  @requires ^Pre_baz(&a, &b, &c)
  // ...
  @ensures ^Post_baz(&a, &b, &c)
  @*/
void baz(int a, double b, struct s *c);
```

But this is unsatisfactory, because specifications would not be informative anymore, and effectively useless in reviews and as code documentation. To avoid this extreme orientation towards the prover, we allow to embed arbitrary Isabelle-/HOL terms of type *bool* (predicates) in specifications via so-called *quotations*, which live in the expr syntactic category. Such terms can be built up from quantifiers over arbitrary types, set comprehensions, infix operators, and anything else that is available in Isabelle/HOL. Quotations are embedded written between ${ and } braces. Of course it is necessary to let such predicates refer to the program state, in particular to values of objects. Anti-quotations serve this purpose: specification expressions can be spliced into quotations by surrounding them with ‘{ and } braces. During the translation of specifications to pure Isabelle/HOL terms, anti-quotations are substituted by their value according to the formal semantics defined in Sec. 5.3. Quotations and anti-quotations may be arbitrarily nested, though in practice only a single level of anti-quotations occurs, which sometimes contains quoted references to bound Isabelle/HOL variables. The following is a postcondition illustrating the use of quotations and anti-quotations.

$$\textbf{@ensures}\ \$\{\mathit{Complex}\ `\{\mathsf{x}+\mathsf{e}\}\ `\{\backslash\mathsf{\mathit{result}}\} \in \{p.\ |p| \leq 1\}\}$$

Assuming that the specified function returns a **double**, and that x and e are both function parameters of same type, the postcondition states that the complex number with real part x + e and with the function's result as imaginary part will have a modulus of at most 1, i. e. the complex number lies within the unit circle. (*Complex* is the constructor of Isabelle's datatype of complex numbers.)

Representation Functions

A particularly common usage pattern of back-references to program entities in quoted Isabelle/HOL terms is as arguments to representation functions. A *representation function* is one that builds a domain value out of the program value(s) forming its representation in the C program. Remember the idea that functions implementing domain operations —as opposed to 'auxiliary' operations related to, e. g., logging, memory management, or data persistence— should be specified in terms of the domain vocabulary as formalised in the Isabelle/HOL domain model. To enable such specifications, program values need to be 'lifted' into the domain somehow. While the memory model of Chapter 4 provides a generic representation of all program values in the formalisation, representation functions instead directly yield proper domain values which are independent of their concrete program representation. There is no single function type subsuming all representation functions, but the general type pattern is

$$R :: \mathit{State} \Rightarrow \mathit{Loc} \Rightarrow \mathit{Val}^n \Rightarrow \alpha$$

for a domain type α, which might be $(\mathit{real} \times \mathit{real})\ \mathit{set}$ for sets of 2D-points, or even a function type like $\mathit{real} \Rightarrow \mathit{real}$. Every such function R expects a program state as its first argument. The C values forming the representation are read in this state, usually starting at the address given by the second argument (*Loc*). Further required program values can be passed (Val^n), for example to provide length information for arrays. (Program values are scalar here: they are either numbers or addresses, cf. Sec. 4.2.2). A simple example of a representation function is one that lifts an array of points of type

```
struct point { double x; double y; } ps[LEN];
```

to a set of mathematical vectors (of type $\mathit{real} \times \mathit{real}$); this function would have the following Isabelle/HOL type:

$$\mathit{PointSet} :: \mathit{State} \Rightarrow \mathit{Loc} \Rightarrow \mathit{int} \Rightarrow (\mathit{real} \times \mathit{real})\ \mathit{set}$$

and it can be used in specification expressions within quotations via the syntax ^PointSet{ps, LEN}; as with state dependent functions, of which representation functions form a subclass, the initial state argument is implicit. This notation for referring to representation functions in quotations is just syntactic sugar, and it is equivalent to the regular anti-quotation ‘{^PointSet(ps, LEN)}. We can also conveniently refer to an evaluation of a representation function in the pre-state within quotations in postconditions via ^^PointSet{ps, LEN}, which decodes to the rather unwieldy expression ‘{\old(^PointSet(ps, LEN))}.

To illustrate how representation functions are used when there is a discrepancy between the program representation type and the domain model type in terms of their respective 'size', i. e. the range of values they represent, we take a look at the robotics domain formalised in the SAMS project. Here, one is interested in a particular class of two-dimensional geometric transformations —i. e., essentially functions $real \times real \Rightarrow real \times real$— that model the motion of physical objects. In the two-dimensional case, we call these transformations *rigid body transforms* (RBT), which are uniquely defined by two parameters: a real value $\phi \in [0, 2\pi)$ for the rotation part, and a vector $(x, y) :: real \times real$ for the translation part. The interval $[0, 2\pi)$ can be described precisely in Isabelle/HOL as $\{r :: real.\ 0 \leq r \wedge r < 2\pi\}$. For smooth use in proofs it is more practical to define the type denoting this interval by the quotient of the set of real numbers (denoted *UNIV* below) over the equivalence relation relating all real numbers whose absolute difference is an integer multiple of 2π.[2]

```
typedef (SO2)
  SO2 = "UNIV // {(a, b). ∃ k::int . a = b + 2 * pi * real k}"
```

Type *SO2* is now isomorphic (though not equal) to $[0, 2\pi)$, and two automatically derived morphisms allow us to switch between the two. Using *SO2* instead of *real* has the obvious benefit that every value uniquely represents a value in the interval. In the case of *real*, 0, 2π, 4π, etc. would either all represent the same value (in an interpretation *modulo* 2π), or several of them would simply be invalid. Getting back to transformations, every rigid body transform can now uniquely be described by a pair of type $RBT = (real \times real) \times SO2$, and a function

$$Tm :: RBT \Rightarrow real \times real \Rightarrow real \times real$$

that transforms a two-dimensional point according to the given RBT.

So much for the domain model; C, on the other hand, does not have such powerful type construction facilities. Furthermore, the choice of datatypes is often influenced by considerations that are not quite as important in a domain formalisation, like the efficiency of execution, the existence of suitable libraries, or the compactness of the representation. Ignoring the latter, rigid body transformations might well be implemented via a general datatype of 3×3 matrices, where for an RBT given by rotation ϕ and translation (x, y) one would use the matrix

$$\begin{pmatrix} \cos(\phi) & -\sin(\phi) & x \\ \sin(\phi) & \cos(\phi) & y \\ 0 & 0 & 1 \end{pmatrix}$$

The concrete C datatype might look as follows:

```
typedef struct matrix3 {
  double v[3][3];
} matrix3_t;
```

This datatype allows several objects that are not rigid body transforms; among others, all matrices in which the last row is not equal to (0 0 1) are not. To express in a specification that a given matrix is an RBT, to which we can

[2] We call it *SO2* to remind ourselves of the fact that the interval is isomorphic to the special orthogonal group $SO(2)$.

```
/*@
  @requires ^is_RBT(m)
  @memory
    src[:num] <*> dst[:num] <*> *m
  @modifies dst[:num]
  @ensures ${ ^PointSet{dst, \result} =
              Tm ^RBT{m} ' ^PointSet{src, \result} }
  @*/
int transform(matrix3_t *m,
              const point_t *src,
              point_t *dst,
              int num);
```

Figure 3.3: Functional specification of a matrix transformation operation: input parameter m must be a rigid body transform, in which case the input points src will be transformed by m, where the result is placed into dst.

safely apply the corresponding representation function lifting it to an element of Isabelle/HOL type *RBT*, we require a *recogniser function* that characterises all RBT matrices. We can define a state dependent function for use in specifications for this purpose, which has an almost identical signature as the corresponding representation function

$$\begin{aligned} \textit{is-RBT} &:: \textit{State} \Rightarrow \textit{Loc} \Rightarrow \textit{bool} \\ \textit{RBT} &:: \textit{State} \Rightarrow \textit{Loc} \Rightarrow \textit{RBT} \end{aligned}$$

The use of these functions is demonstrated in Fig. 3.3: C function transform is supposed to transform the array of points src of length num according to matrix m and put the result into dst. The precondition is that m represents an RBT, ensured by the state dependent function *is-RBT*. The postcondition is formulated exclusively inside the domain: dst gets interpreted as a set of real points, up to and not including its \result's index (implying that callers of this function will have to check the result in order to know how many points were actually copied). This set is equal to the set of points obtained by transforming the points represented by src (again, up to \result) via *Tm* and according to the rigid body transform obtained from m.

From the specifier's point of view, it is only necessary to know which representation functions and representation recognisers exist, respectively which ones he requires, so that they can be integrated into the domain formalisation. That is to say, he can see the collection of recognisers and representation functions as a library for use in specifications. From the verifier's point of view it is necessary to know what properties these functions have, in particular with respect to dependencies on memory locations, so that the right simplification procedures can be applied during verification condition generation. This issue is discussed in Sec. 6.5.

3.3.3 Types in Specification Expressions

In this section we illustrate some properties of C's built-in datatypes to motivate our decision to use a simpler type system for specification expressions. The

C standard defines 14 so-called *basic types* which together with enumerations form the *arithmetic types* (cf. Fig. 3.4). They fall into two categories: (real) floating types and integer types. Restrictions are placed on the possible value ranges that each type covers, and there exists an ordering between the unsigned integer types as well as the signed ones: a value of type **signed char** must always fit into a **short int**, which must fit into an **int**, etc. However, the standard does not mandate the exact bit-width or value range of any type except the character types. The characteristics of floating-point values are also *implementation defined.* In particular, the accuracy of floating-point operations as well as the existence of special values like infinity or *NaN* (not-a-number) are not guaranteed.

Arithmetic Conversions

The part of the language definition concerned with implicit and explicit conversions between arithmetic types is a little intricate. Conversions from any integer type to an unsigned integer type always succeed, with conversions to smaller types effectively being defined by bit truncation, while the result of conversions to smaller signed integer types may differ between implementations (the behaviour is, again, implementation-defined). Converting a value of type **double** that is larger than any **float** into the latter type results in undefined behaviour. And due to the so-called *integer promotion* rules arithmetic operations (addition etc.) of integer values of types smaller than **int** are actually always performed in type **int**. On a common architecture where the lengths of **short int**, **int** and **long long int** are 16, 32, and 64 bits, respectively, this results in the peculiar fact that the addition of two **short int**s can never overflow when assigned to an **int**, while the addition of two **int**s might well overflow, resulting in undefined behaviour, even when the result is assigned to a **long long int**.

A Type System for Specification Expressions

It is unsatisfactory to be forced to deal with such numerical intricacies within specifications. If a particular specification is expressly concerned with numerical properties of a function, then one obviously needs ways to express the overflow behaviour, conditions on the definedness of operations, ranges of arithmetic types, and so forth. In most cases, however, a specification is concerned with more abstract, *functional* properties and in those cases it is more convenient, arguably even necessary, to be able to abstract away from numerical details. Our solution is to interpret all specification expressions over (unbounded) mathematical integers, real numbers, booleans and, thanks to the quotation mechanism, *any* other Isabelle/HOL type. This leads to the following four *basic types of specification expressions*:

Type	*Isabelle/HOL equivalent*	*Math. domain*
_Bool	*bool*	$\{True, False\}$
_Int	*int*	$\mathbb{Z}$
_Real	*real*	$\mathbb{R}$
_Any	⟨*any type*⟩	?

Type _Any can be regarded as a radically simple way to reconcile the fundamentally different type systems of Isabelle/HOL (which is founded on the polymorphic Hindley-Milner type system, augmented with ad-hoc polymorphism in

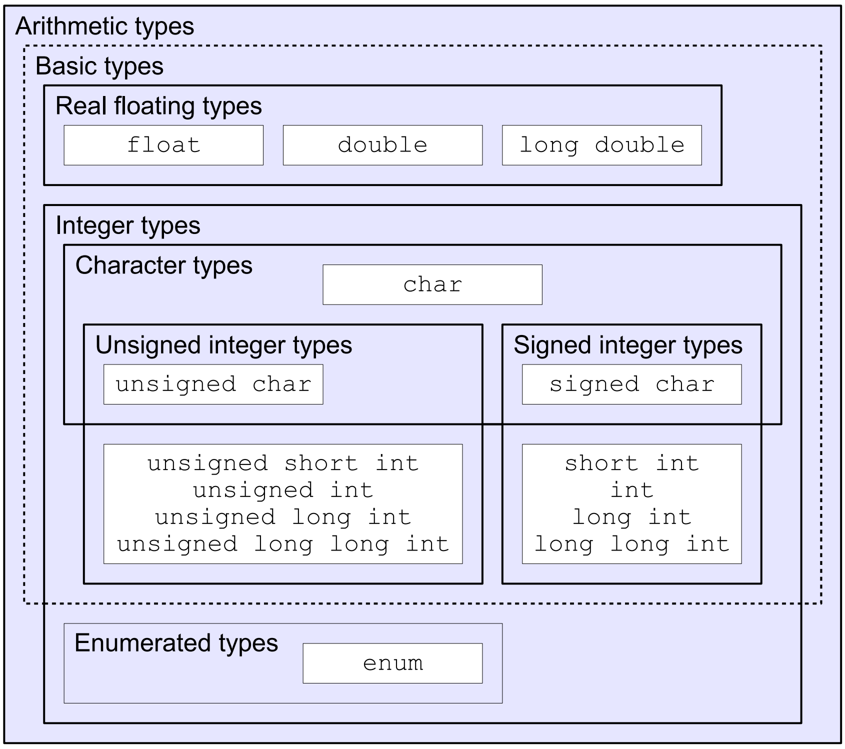


Figure 3.4: The C type taxonomy. Enumerated types are not included in the CSI subset of C; _Bool, _Int, and _Real are also regarded as *arithmetic types.*

the spirit of Haskell's type classes [116]) with the type system of C. The idea is that every Isabelle/HOL type except those three listed in the table above is assigned the type _Any. In the type system of specification expressions, _Any becomes the super-type of all other types. That is, wherever a value of type _Any is expected, one can pass a value of any other type. From C, we inherit structured types, pointer types, and array types. The concepts of type qualifiers, storage classes, incomplete types (e. g., arrays without a specified element count), and union types do not exist in specification expressions. It is convenient to include function types for giving types to Isabelle/HOL functions embedded via $-quotations etc. Therefore, types are defined by the following abstract grammar:

$$\begin{array}{rcll} \tau & ::= & \textit{basic-type} & \\ & | & \tau\ * & \text{(pointer)} \\ & | & \tau[N] & \text{(array)} \\ & | & \textit{tag}\ \{\ f_1 : \tau_1;\ \ldots\ ;\ f_n : \tau_n\ \} & \text{(struct)} \\ & | & \tau \Leftarrow (\tau_1, \ldots, \tau_n) & \text{(function)} \end{array} \tag{3.1}$$

We require all lvalues occurring in specification expressions as defined by the abstract grammar in Fig. 3.2 to be basic types or pointers: structured or even function values are not allowed. As in C, an lvalue of array type used in an expression 'decays' into its corresponding pointer type. We elide a formal definition of the translation of C lvalue types to those of Eq. (3.1), and appeal to the reader's intuition. The sole noteworthy fact is that all C integer types are converted to _Int, while all floating types are converted to _Real. For example, in the context of a declaration **struct** s { **int** a; **double** b[3]; } s1; the translated type of s1 is s { a : _Int; b : _Real[3] }, while s1.b has type _Real $*$ in an expression, due to array type decay. Type definitions via **typedef** are handled transparently as in C, i. e. the defined type is considered equal to the defining type and no explicit conversion between such types is necessary to achieve type compatibility.

The type system of specification expressions is presented in Fig. 3.5. The presentation is standard: an expression e is well-typed with type t if the typing judgment $\Gamma \vdash e : t$ can be derived via the given rules. The type environment Γ is used to map lvalues (instead of plain variables) to their respective types. We assume the initial type environment of a specification expression to contain all identifiers that are visible within the expression. These comprise global variables, function parameters, declared Isabelle/HOL identifiers and constants, as well as abbreviations (cf. Sec. 3.5.3). The first rules are concerned with lvalues, literals and arithmetic operations. Rules (Not), (Comp) and (Conn) show that the boolean type is taken seriously: in contrast to C, where **int** is the result type of comparisons, conjunctions, disjunctions and negation, in specifications these all yield and, where appropriate, expect operands of type _Bool. Following rule (IfThenElse), rules (Conv1) and (Conv2) specify that _Int can be implicitly converted to _Real, which itself can be converted to _Any. Other conversions must be realised through the use of quoted Isabelle/HOL functions: imagine a function *roundToNearest* :: *real* $\Rightarrow$ *int* emulating the semantics of a cast from, e. g., **double** to **int** which can be referenced as $roundToNearest(x) for some x of type _Real. It would be possible to allow explicit casts in specification expressions and to give them an implementation defined semantics by parameterising the formalisation over the architecture at hand. However, we prefer to require an explicit reference to the Isabelle/HOL function defining the desired conversion. (It is rather improbable that specifications which rely on implementation defined behaviour will look identical for different architectures, which casts doubt on the benefit of a parameterisation.) In the scope of quantifiers, the type environment is extended by the bound identifier and its type, which is converted to _Real or _Int as done with regular lvalues. Hence, the environment for typing expression E in \forall **char** c; E is extended by (c $\mapsto$ _Int). The type rules for \old, @label, and memory predicates are trivial. In rules (Dollar) and (Hat), the type environment is expected to yield the function type for the respective Isabelle/HOL function. Note that the implicit state argument of 'hat'-referenced functions is also hidden in its type. (Quote) expresses quotations are of boolean type and that all anti-quotations must be well-typed, whereas no restrictions are placed upon 'raw' Isabelle/HOL text.

3.4 Memory Layout Descriptions

It is well known that one of the most severe complications both for the analysis and the verification of imperative programs is caused by aliasing[114]. Alias-

$$\frac{\Gamma(\textsf{lval}) = t}{\Gamma \vdash \textsf{lval} : t} \qquad \frac{\Gamma \vdash \textsf{lval} : t}{\Gamma \vdash \textsf{\&lval} : t \,*}$$

$$\frac{}{\Gamma \vdash \backslash\textsf{true} : \textsf{\_Bool}} \qquad \frac{}{\Gamma \vdash \backslash\textsf{false} : \textsf{\_Bool}}$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 \oplus e_2 : t} \qquad \begin{array}{c} \oplus \in \{+, -, *, /\} \\ t \in \{\textsf{\_Int}, \textsf{\_Real}\} \end{array}$$

$$\frac{\Gamma \vdash e_1 : \textsf{\_Int} \quad \Gamma \vdash e_2 : \textsf{\_Int}}{\Gamma \vdash e_1 \ \textsf{\%}\ e_2 : \textsf{\_Int}}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash -e : t} \ t \in \{\textsf{\_Int}, \textsf{\_Real}\} \qquad \frac{\Gamma \vdash e : \textsf{\_Bool}}{\Gamma \vdash \,!\,e : \textsf{\_Bool}} \ (\textsc{Not})$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 \lhd e_2 : \textsf{\_Bool}} \qquad \begin{array}{c} \lhd \in \{<, <=, >, >=, ==, !=\} \\ t \in \{\textsf{\_Int}, \textsf{\_Real}\} \end{array} \qquad (\textsc{Comp})$$

$$\frac{\Gamma \vdash e_1 : \textsf{\_Bool} \quad \Gamma \vdash e_2 : \textsf{\_Bool}}{\Gamma \vdash e_1 \Diamond e_2 : \textsf{\_Bool}} \quad \Diamond \in \{\textsf{\&\&},\ \textsf{||},\ \textsf{-->}, \textsf{<->}\}$$

$$(\textsc{Conn})$$

$$\frac{\Gamma \vdash b : \textsf{\_Bool} \quad \Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \backslash\textsf{if}\ b\ \backslash\textsf{then}\ e_1\ \backslash\textsf{else}\ e_2 : t} \quad (\textsc{IfThenElse})$$

$$\frac{\Gamma \vdash e : \textsf{\_Int}}{\Gamma \vdash e : \textsf{\_Real}} \ (\textsc{Conv1}) \qquad \frac{\Gamma \vdash e : \textsf{\_Real}}{\Gamma \vdash e : \textsf{\_Any}} \ (\textsc{Conv2})$$

$$\frac{\Gamma(\textsf{x} \mapsto t) \vdash e : \textsf{\_Bool}}{\Gamma \vdash \backslash\textsf{forall}\ t\ \textsf{x};\ e : \textsf{\_Bool}} \qquad \frac{\Gamma(\textsf{x} \mapsto t) \vdash e : \textsf{\_Bool}}{\Gamma \vdash \backslash\textsf{exists}\ t\ \textsf{x};\ e : \textsf{\_Bool}}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \backslash\textsf{old}\ e : t} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash e\ \textsf{@ label} : t}$$

$$\frac{\Gamma \vdash e : t\,*}{\Gamma \vdash \backslash\textsf{valid}(e) : \textsf{\_Bool}} \qquad \frac{\Gamma \vdash e_1 : t\,* \quad \Gamma \vdash e_2 : \textsf{\_Int}}{\Gamma \vdash \backslash\textsf{array}(e_1, e_2) : \textsf{\_Bool}}$$

$$\frac{\Gamma \vdash e_1 : t_1\,* \quad \Gamma \vdash e_2 : t_2\,*}{\Gamma \vdash \backslash\textsf{unrelated}(e_1, e_2) : \textsf{\_Bool}} \qquad \frac{\Gamma \vdash e_{\{1,3\}} : t_{\{1,3\}}\,* \quad \Gamma \vdash e_{\{2,4\}} : \textsf{\_Int}}{\Gamma \vdash \backslash\textsf{separated}(e_1, e_2, e_3, e_4) : \textsf{\_Bool}}$$

$$\frac{\Gamma(\textsf{ident}) = t \Leftarrow (t_1, \ldots, t_n) \quad \Gamma \vdash e_i : t_i}{\Gamma \vdash \textsf{\$ident}(e_1, \ldots, e_n) : t} \quad (\textsc{Dollar})$$

$$\frac{\Gamma(\textsf{ident}) = t \Leftarrow (t_1, \ldots, t_n) \quad \Gamma \vdash e_i : t_i}{\Gamma \vdash \textsf{\^{}ident}(e_1, \ldots, e_n) : t} \quad (\textsc{Hat})$$

$$\frac{\Gamma \vdash e_i : t_i\ (e_i\ \textit{not raw Isabelle/HOL text})}{\Gamma \vdash \textsf{\$\{}\ e_1\ \cdots\ e_n\ \textsf{\}} : \textsf{\_Bool}} \quad (\textsc{Quote})$$

Figure 3.5: The type system of specification expressions

ing concerns the fact that an object in memory may be referred to by several, possibly unrelated identifiers or expressions. In the case of C, and using the terminology of Bornat [29], there are four possible forms of aliasing. They are *subscript aliasing*, where a[E] and a[F] will refer to the same memory location when E and F evaluate to the same integer value; *overlap aliasing* where the assignment to a whole **struct** affects the value of its members; *view aliasing*, which occurs when different fields of a **union** are used to access the same memory area; and, most importantly, *pointer aliasing* concerns the situation where either two pointers contain the same or related addresses, or where a pointer contains the address of an object also denoted by a variable. Since the language subset supported by CSI precludes assignment to whole structures and the use of unions, we only need to deal with subscript and pointer aliasing.

From the viewpoint of function specifications, the most common case of aliasing involves pointer-valued function parameters, when either addresses of arrays are passed, or when call-by-reference is simulated by passing the addresses of single objects. The following function foo illustrates both cases: its argument p1 is expected to be a pointer to a single read-only **struct** s which might itself be a local object of the calling function, or a global object. p2 shall contain the address of an array of **struct** s objects of length p2_len, and out is an output parameter containing a valid address, again for a single object.

```
void foo(const struct s *p1,
         struct s *p2,
         int p2_len,
         struct s *out);
```

Without any restrictions on the possible aliasing between these pointers, the specification —and particularly the verification— of this function will be very complex, since all of *p1, p2[i] ($0 \leq$ i $<$ p2_len) and *out might change their values when *out is modified inside the function. (That p1 is declared as pointer to constant is irrelevant, as it does not force the object pointed to to be constant.) In most practical cases, functions like this are, however, only called with certain aliasing patterns that exhibit well-defined and well-specifiable behaviour. For example, for the standard function

```
void *memcpy(void *s1, const void *s2, size_t n)
```

it is expressly stated that “behaviour is undefined, if s1 and s2 overlap”[81], implying that the function should not be used for overlapping memory areas. In the foo example, one might imagine that p1 and p2 are only used for reading, while out is written to. One could thus demand that out is aliased neither to p1 nor to any p2[i].

Furthermore, the concept of arrays in C is a very weak one. Arrays are just contiguous blocks of memory, with the block size determined by the size of the array element type. Array variables ‘decay’ to pointers of their first element’s address when used in expressions, so that passing an array to a function cannot be distinguished from passing the address of a single object. Information like the capacity or the current element count of an array needs to be maintained externally by the programmer. Typically, when arrays are passed to functions, an additional parameter is used containing the minimum size of the array. The function’s declaration does not formally reveal this fact, so that informal naming schemes are often used (like p2 and p2_len above). It is therefore desirable to be able to express the relationship between two such variables and, generally, to specify the required capacity of an array in function interface specifications.

Additionally, when dereferencing a pointer (whether to an array or to a single object), it must contain a *valid address*, i. e. an address at which an object of the type given by the pointer is actually stored. There are three ways to violate this property: a pointer might contain the `NULL` value, it might not have been initialised and hence contain an arbitrary address, or it might have been assigned an invalid value. Functions that unconditionally dereference pointers require a precondition that ensures their validity.

CSI provides predicates for use in pre-/postconditions as well as a particular notation to be used `@memory` annotations, which enable the specification of allowed aliasing patterns for function interfaces and of requirements on pointer and array validity. These are discussed in the next sections.

3.4.1 Memory Descriptors

We considered two ways of expressing that memory areas are unrelated (i. e., do not overlap). One way is to express this as a property of pointer values and types, e. g. with a special predicate `\unrelated`. The intended meaning of `\unrelated(p, q)` (with `p` and `q` of pointer type) would be that the area of size **sizeof**`(*p)` starting at `p` does not overlap the corresponding area of `q`. In the case of arrays, a predicate `\separated(a, n, b, m)` specifies that `a` and `b` as arrays of sizes `n` and `m` do not overlap. We can express arbitrary aliasing situations between pointers and arrays of fixed size through boolean combinations of such binary specifications of unrelatedness. However, pairwise specifications are rather inelegant as they lead to a quadratic blowup of the specification in the common situation where all or most pointers need to be unrelated. The expression `\unrelated(a, b) && \unrelated(b, c) && \unrelated(c, a)` illustrates the problem. Nonetheless, the operators `\separated` and `\unrelated` are part of CSI and can be used in pre-/postconditions.

Mvalues For a particularly common class of specifications, in which the precondition requires exactly one pattern of memory layout and pointer/array validity, this pattern can be written concisely as a `@memory` annotation. Memory areas are here described via *mvalues*, which can be regarded as a modification of the concept of C lvalues, and two binary combinators over these. The intuition is that an mvalue denotes collections of objects and hence describes the memory areas occupied by them, while the two combinators express separation constraints and unions of memory areas. The abstract syntax of mvalues is described in Fig. 3.6. An identifier mvalue `m` denotes the object referred to by the variable of that name and describes its memory area, of size **sizeof**`(m)`. A dereferenced mvalue `*m` denotes the objects obtained by interpreting the objects denoted by `m` as pointers, and dereferencing them. The array range expression `m[i:j]` denotes all objects of the arrays denoted by `m`, from index `i` up to and including `j−1`. `m.f` denotes all `f`-fields of the objects denoted by `m`. The formal semantics of mvalues in Isabelle/HOL are given in Sec. 5.5. The type rules for expressions extend to mvalues in the obvious way. Some syntactic sugar is served: `a−>f` means `(*a).f`, `v[:n]` is the same as `v[0:n]`, and `v[i]` is equivalent to `v[i:i+1]`.

$$
\begin{array}{lcll}
\mathit{mval} & ::= & \mathit{ident} & \text{(variable)} \\
 & | & *\mathit{mval} & \text{(pointer dereference)} \\
 & | & \mathit{mval}[\mathit{triv\text{-}expr} : \mathit{triv\text{-}expr}] & \text{(array range)} \\
 & | & \mathit{mval}\,.\,\mathit{ident} & \text{(field selection)} \\
 & & & \\
\mathit{triv\text{-}expr} & ::= & \mathit{ident} & \\
 & | & \mathit{int\text{-}literal} & \\
 & | & \mathit{triv\text{-}expr} \otimes \mathit{triv\text{-}expr} & \\
 & & \quad \otimes \in \{+, -, *, /, \%\} &
\end{array}
$$

Figure 3.6: Abstract syntax of memory descriptors (*mvalues*)

Mvalues allow us to describe a memory area whose size is determined either statically (e. g., `*p` has a fixed size given by the type of `p`), or, in the case of arrays, is given by the values of integer variables (as in `a[:len]`). We cannot describe graph-shaped structures that arise from arbitrary repetitions of dereferencing and recursive data structures; the typical example are lists of unbounded length. This, however, is not a problem in our application domain, as memory allocation is prohibited (see Sec. 5.1), which practically precludes the use of dynamically-sized datatypes.

3.4.2 Separation Constraints

We can now combine mvalues with two operators to form larger memory areas and to express separation constraints. The operator <+> conjoins two memory areas and (intuitively) yields the memory area described by its operands. The second operator <*> also conjoins two memory areas, but additionally requires that the areas of its operands do not overlap, i. e. do not have a single memory location in common. We can build terms over mvalues with these combinators to form memory layout descriptions that express both the validity of the involved memory areas as well as their separation. As an example, we state the memory requirements on `foo` from above as follows:

```
@memory (*p1 <+> p2[:ps_len]) <*> *out
```

This is equivalent to the following precondition (merely indicating that validity assertions are derived from a memory layout description, too):

```
@requires \unrelated(p1, out)
  && \separated(p2, ps_len, out, 1)
  && \valid{...}
```

One particularly powerful feature of combinator-based expressions is that they allow a linear formulation of pairwise separation constraints, as in

```
@memory *a <*> *b <*> c[:n] <*> d
```

which concisely captures the separation of four memory areas.

We have formalised memory layout descriptions in Isabelle, and proved the important property that memory layout descriptions can be translated to a conjunction of binary separation constraints, as definable by the `\unrelated` and `\separated` predicates. We increase generality by defining our theory over arbitrary sets instead of memory areas. We then instantiate the theory for the concrete memory model used (cf. Sec. 4). The datatype for memory layout descriptions has a base constructor representing a concrete set of locations, and two constructors representing the combinators, where we use the typographically more appealing symbols ◊ and ⊕ for <*> and <+>, respectively.

```
datatype 'a mem_descr = L "'a set"
  | Sep "'a mem_descr" "'a mem_descr"    (infixl "◊" 80)
  | Join "'a mem_descr" "'a mem_descr"   (infixl "⊕" 90)
```

We provide semantics by defining when a state satisfies a memory description. States are abstracted to sets of locations, to be interpreted as those locations, i. e. that part of memory, at which the state is defined. For example, if states are partial functions from addresses to bytes: $S : nat \rightharpoonup word8$, then we are interested in the state's domain $dom(S)$. A state satisfies an atomic memory layout description if it contains all locations mentioned in the description. The join operator ⊕ is simply interpreted as a conjunction. To satisfy `M ◊ N`, it must be possible to split the state into two disjoint parts that each satisfy one of the operands. This neatly captures the separation requirement that `M` and `N` may not have locations in common.

```
fun sat_mem_descr :: "'a set ⇒ 'a mem_descr ⇒ bool"
  (infix "⊨" 70)
where
"S ⊨ L X = (X ⊆ S)"  |
"S ⊨ M ⊕ N = (S ⊨ M ∧ S ⊨ N)" |
"S ⊨ M ◊ N = (∃S' S''. S = S' ∪ S'' ∧
                          S' ∩ S'' = {} ∧ S' ⊨ M ∧ S'' ⊨ N)"
```

One might have expected a similar definition for ⊕ as was given for ◊, except for the disjointness condition. However, we observe that satisfaction of memory layout descriptions is monotone, and hence the two definitions are equivalent:

```
lemma sat_mono:
  "∀S S'. S ⊨ M ⟶ S ⊆ S' ⟶ S' ⊨ M"
```

```
lemma Join_decomposition:
  "S ⊨ M ⊕ N = (∃S' S''. S = S' ∪ S'' ∧ S' ⊨ M ∧ S'' ⊨ N)"
```

From the definitions it is obvious that both ◊ and ⊕ are associative and commutative:

```
lemma Sep_assoc: "S ⊨ (A ◊ B) ◊ C = S ⊨ A ◊ (B ◊ C) "
lemma Sep_commute: "(S ⊨ M ◊ N) = (S ⊨ N ◊ M)"
lemma Join_assoc: "S ⊨ (A ⊕ B) ⊕ C = S ⊨ A ⊕ (B ⊕ C)"
lemma Join_commute: "(S ⊨ M ⊕ N) = (S ⊨ N ⊕ M)"
```

To normalise every memory layout description into a ⊕-combined sequence of binary separation expressions of the form `L X ◊ L Y`, we require a lemma about distributivity of ◊ over ⊕ and one that allows us to split up terms like `M ◊ N ◊ O`. These lemmas depend on the following crucial state intersection lemma:

lemma *sat_Inter:*
```
  "∀S S'. S ⊨ N ⟶ S' ⊨ N ⟶ (S ∩ S') ⊨ N"
```

Proof. By induction over the structure of *N*. The interesting case is *N = N1 ◊ N2*: we may assume the proposition both for *N1* and *N2* and need to show *(S ∩ S') ⊨ N1 ◊ N2*, given that *S* and *S'* both satisfy *N1 ◊ N2*. So we can split $S = S_1 \uplus S_2$ and $S' = S'_1 \uplus S'_2$. From the assumptions we obtain $(S_1 \cap S'_1) \models N1$ and $(S_2 \cap S'_2) \models N2$. We take the trivially disjoint split $S \cap S' = (S_1 \cap S'_1) \uplus ((S_1 \cup S_2) \cap (S'_1 \cup S'_2) - (S_1 \cap S'_1))$ and using *sat_mono* only need to show $S_2 \cap S'_2 \subseteq (S_1 \cup S_2) \cap (S'_1 \cup S'_2) - S_1 \cap S'_1$, which is a set theoretic validity. □

lemma *Sep_Join_distrib:*
```
  "(S ⊨ (M ⊕ M') ◊ N) = (S ⊨ (M ◊ N) ⊕ (M' ◊ N))"
```

lemma *Sep3_Join:*
```
 "(S ⊨ A ◊ B ◊ C) = (S ⊨ (A ◊ B) ⊕ (B ◊ C) ⊕ (C ◊ A))"
```

Proof. In both cases, we explicitly construct the intuitively appropriate splits and make use of *sat_Inter* and *sat_mono* to narrow down or extend states as necessary. □

Memory layout descriptions are inspired by the concept of the separating conjunction operator defined in separation logic [132]. Instead of integrating the operator (typically denoted by the symbol $*$ in the literature) into the logic and putting it on one level with the usual logical connectives such as $\wedge$ and $\vee$, we took a more lightweight approach and only apply ⊕ and ◊ to memory layout descriptions. We thereby stay in the well-understood realms of Isabelle's higher-order logic with good support for proof automation. The distinction becomes clear in a simple example: the separation logic formula $x \mapsto 1 * y \mapsto 2$ expresses that the memory location denoted by x contains the value 1 *and* is separated from the memory location denoted by y, containing the value 2. Properties of memory values are intertwined with properties of memory structure. On the other hand, the memory layout description $L\ \{x\}\ \lozenge\ L\ \{y\}$ is a purely structural description disregarding any values. In particular, the operator's argument $L\ \{x\}$ is not a formula, in contrast to $x \mapsto 1$.

Independently from our work, Gast [62] has given a formalisation of memory layouts similar to ours. His focus is on proving properties of programs written in an imperative toy language performing low-level memory operations, particularly in the presence of dynamic memory. There is no analog of the ⊕ operator, as he does not aim for concise descriptions of memory layouts, i. e. readable specifications.

3.4.3 Validity of References and Arrays

We said that memory layout descriptions implicitly express the validity of the mvalues involved and referred to the Isabelle formalisation (Sec. 5.5) for semantics. When @memory annotations are not used to describe memory validity, e. g.

when the respective memory cannot be described by mvalues because it has a triangular instead of a quadratic shape (see below), two special predicates can be used in pre-/postconditions to express the validity of pointers and arrays. \valid(p) evaluates to true if the pointer p, whose type we here assume to be T *, contains an address of an object of type T. Furthermore, \array(−, −) is a binary predicate expecting as first argument a value of pointer type and an integer value as second argument. \array(a, n) evaluates to true if a points to an element of an array, such that all a[0] to a[n−1] are valid accesses into the array. For example, in the context of an array of **int** pointers, **int** *a[N], the following expression requires that every pointer in the array points to an **int** array at least the size of the corresponding pointer array index i.

```
\forall int i; 0 < i < N —> \array(a[i], i)
```

Such a ‘triangular’ shape cannot be specified through an mvalue, and hence in admittedly rare situations like this one has to resort to using validity predicates.

3.5 Further Language Elements

So far we have concentrated on annotations used for functional specifications. A few more technical annotations are necessary to enable the type checking and the formal verification of these specifications.

3.5.1 Modification Frames

In addition to pre- and postconditions, Fig. 3.1 and Fig. 3.3 already contained a *modification frame*, or @modifies clause. A modification frame is described by a sequence of mvalues (cf. Sec. 3.4.1) or the special symbol \nothing, in case the function does not modify any externally visible variables, but only its local variables. It specifies which memory locations are *possibly and at most* modified by an arbitrary execution of the function starting in a program state that satisfies the precondition. In other words, every memory location not denoted by any of the mvalues will have the same value before and after execution of the function. The mvalues comprising a modification frame are evaluated in the pre-state.

The importance (and difficulty) of specifying modification frames has been studied in the literature many times (Borgida et al. [28] provide an in-depth discussion) and their relevance is not restricted to program specifications, but extends to the field of artificial intelligence, where the problem is known as the *frame problem.* Summarising the possible changes in memory caused by the execution of a function is particularly relevant in all places where that function is called: both the programmer and the verifier have to be able to derive which variables will retain their value and which properties will remain true across the function call. One might argue that a modification frame could simply be made a part of the postcondition: to state that variable a maintains its value across the execution of function foo, simply include the expression a == \old a in foo’s postcondition. This approach, however, does obviously not scale to realistic programs with hundreds and thousands of memory locations. The huge advantage of specifying what *is* (possibly) changed, as opposed to being explicit about what is *not* changed, is that the former is a local property: the modification frame can be determined by looking at the function body and the modification frames in the interface specifications of all called functions. To specify what is not changed, one needs to be able to name, i. e. to reference, all entities existing in a program.

Modification frames have an over-approximative character. The effect of a function execution on memory can always be made more precise inside the postcondition. Consider the following code snippet:

```
/*@
  @modifies a
  @ensures
    \if c <= 0 \then a == \old a \else a == 0
  @*/
void reset_if_pos(int c) {
  if (c > 0) {
    a = 0;
  }
}
```

3.5.2 Statement Annotations

In contrast to function specifications, which actually *prescribe* the behaviour of their corresponding functions, statement annotations merely *describe* the effects of statements. Their inclusion does not alter the semantics of program execution nor the specification of the function body in which it is contained. Statement annotations serve two purposes: (a) they document the effects of statements in a precise manner. This makes it possible to understand the code in terms of the domain model on an even finer grained level than that of function interfaces. (b) They are used during the automatic derivation of verification conditions in Isabelle/HOL. It is in particular necessary to provide invariant annotations for each looping statement in a function to enable the fully automatic derivation of verification conditions.

Join Arbitrary statements can be annotated with a @join annotation that may be accompanied by a @modifies clause. The purpose of such an annotation is to provide a concise summary of the effects of the statement. It is useful in cases where all program paths contained in a compound statement (some of them possibly empty) establish a common property. Such an annotation thereby effectively *joins* these program paths into one during verification, avoiding a blow-up in the size of verification conditions. The following code snippet gives an example, where a local variable a is set to its absolute value. This could of course be achieved by an appropriate call to the abs() function; but imagine that this had to be avoided for performance reasons.

```
/*@
  @join a == $abs(\old(a))
  @modifies a
  @*/
if (a < 0) {
  a = -a;
}
```

The effect of this annotation is that during verification, all statements *following* the **if** will only have to be considered once, under the assumption that both branches of the **if** make the @join predicate true. Had the @join been omitted, subsequent statements would have to be considered twice: for the case where the then branch has been taken *and* for the **else** branch.

The \old operator may be used in @join annotations and therein refers to the value of its operand before execution of the annotated statement. The overall @join expression is evaluated in the state after execution of that statement.

Invariants The second statement annotation concerns the well-known specification of *invariants* of looping constructs (**for**, **do** ... **while**, **while**). Loop annotations must include an @invariant and a @variant annotation and may additionally contain a @modifies clause. This use of invariants is standard, which is why we keep the discussion short. The variant must be an expression of type _Int. It provides a hint to the verification condition generator why a particular loop will terminate. To prove termination of a loop, one generally has to find a well-founded ordering on program states such that an iteration of the loop transforms every program state which makes the loop condition evaluate to true into one that is strictly smaller according to the ordering. Since the ordering is well-founded, this process cannot continue forever, and the loop has to terminate. We take a simple approach in which the ordering on states is expressed through an expression of type _Int, which has to be non-negative in all states satisfying the invariant.

The invariant itself is a specification expression. It can refer to all variables also visible in pre-/postconditions, and additionally to the function's local variables that are in scope at the occurrence of the annotated loop. Due to possible side-effects in loop conditions (as in **while** (++i < n) { ... }) and the existence of three different looping constructs, it is not immediately clear which program states they refer to, i. e. in which states invariants are evaluated. Again, a formal definition can be found in Sec. 6.3, but we provide an intuition here:

- For **while**, the invariant refers to the program state *before* evaluation of the loop condition. This includes both the state before entry into the loop, and all states reached after execution of the loop's body, right before reevaluation of the loop condition. Side-effects in loop conditions thus make it possible that an invariant does not hold *after* the execution of a loop, but only right before the final evaluation of the loop condition.

- Invariants of **for** and **do** ... **while** loops are best explained by a translation to their **while** equivalents, expressed in terms of the C grammar:

  for ($expression^1_{opt}$; $expression^2_{opt}$; $expression^3_{opt}$) *statement*

  and

  do *statement* **while** (*expression*) ;

  are translated to

  { $expression^1_{opt}$; **while** ($expression^2_{opt}$) { *statement* ; $expression^3_{opt}$; } }

  and

  { *statement* ; **while** (*expression*) *statement* }

> Invariants are evaluated exactly as if they were attached to the **while** statement in the ‘expanded’ form. In particular, on entry to a **do** ... **while** loop, the invariant does not need to hold.

The following is a complete loop invariant of a **for** loop implementing multiplication via iterated addition.

```
prod = 0;
/*@
  @invariant 0 <= i && i <= a
    && a * b == (a - i) * b + prod
  @modifies prod, i
  @variant a - i
  @*/
for (i = 0; i < a; ++i) {
  prod += b;
}
```

An invariant can refer to the value of an expression at a labelled statement that has been executed before loop entry via the e @label syntax. It may not use the \old operator.

3.5.3 Declarations and Symbolic Constants

There are four more annotations, that are not attached to particular functions, but rather appear at the file level. They are not concerned with functional specifications, but are used to enable the type checking of specification expressions, to cope with C preprocessor macros, or to avoid the tedious repetition of often used expressions. We describe them in the following paragraphs.

Theory Imports A @theory annotation occurring at file level, i. e. at the level of external declarations, simply advises the front-end performing the translation from C source code to Isabelle/HOL theories to import the specified theories into the theory generated for the C file in which the annotation occurs. For example, @theory SAMSDomain declares that the domain theory of the SAMS project (in file *SAMSDomain.thy* in an appropriate path) should be imported.

Declaration of Isabelle/HOL Functions To perform type checking of specification expressions, it is necessary to assign types not only to program variables and lvalues, but also to every Isabelle/HOL function f that is referenced either as `$f (..)` or `^f (..)`, as rules (Dollar) and (Hat) of Fig. 3.5 indicate. While the types of lvalues are derived from the types given in the respective variable declarations in the C program, a special declaration annotation is used to declare the types of such logic functions. The declaration syntax is that of C function declarations, but the return type must be one of _Real, _Int, _Bool, or _Any, as these are the expression types reflecting the types of the prover language. The types of parameters are declared in the type system of specification expressions; recall that structured types and type definitions retain their C names, but integer and floating-point types are all replaced by _Int and _Real, respectively.

It is also possible to declare ‘nullary’ functions, i. e. constants. Fig. 3.7 depicts some examples in concrete syntax, including declarations of representation

```
/*@theory T @*/

/*@
$function {
  _Real pi;
  _Real sin(_Real x);
}

^function {
  _Bool is_RBT(matrix3_t *m);
  _Any RBT(matrix3_t *m);
  _Any PointSet(point_t *v, _Int len);
}
@*/
```

Figure 3.7: Declaring theory references and state independent as well as state dependent Isabelle/HOL functions.

```
/*@
  ::abbreviation {
    _Bool config_OK(config_t *c) =
      c->status == STATUS_OK &&
      c->x1 < 0.0 &&
      (c->strict == 1 --> c->fail_on_warnings == 1)
  }
@*/
```

Figure 3.8: A naming mechanism for often used specification expressions is provided via the definition of abbreviations.

```
#ifdef VERIFY_MODE
/*@define _Int LEN = 4 * 1024; @*/
#else
#define LEN (4 * 1024)
#endif
```

Figure 3.9: A pattern to redirect C preprocessor definitions to Isabelle/HOL constants for verification purposes.

```
#define LEN (4 * 1024) \
  /*@define _Int LEN = 4 * 1024; */
```

Figure 3.10: Using a postfix annotation to associate expressions with Isabelle-/HOL constants for verification purposes. This only works if the preprocessor supports passing comments through, as with GNU `cpp`'s `-CC` option.

functions and recognisers as well as the declaration of the mathematical constant π, also available in the prover under the name *pi*.

Defining Named Specification Expressions While the domain-related parts of specifications are assumed to use the definitions of the formalisation to express the desired properties, there will always be a need for code-related specifications. Examples are the requirement that certain variables of type **int** must be negative, or that the members of a structure representing some configuration information are consistent. Consistency might include the fact that whenever the .strict member of the structure is set to 1, the .fail_on_warnings member must also be set to 1. Properties like these can sometimes be more conveniently stated in terms of a specification expression rather than as a domain predicate. This holds true particularly for those expressions that are concerned mainly with values of program variables which do not have a direct correspondence in the domain, like for example program configuration information. It is possible introduce names for specification expressions at the file level via an :: abbreviation annotation. In Fig. 3.8 we define the name config_OK for a given specification expression. The expression is parameterised over the configuration pointer c of type config_t *; we use the syntax of C initialised declarations here: abbreviations are declared like C functions with an initialiser. Parameterless abbreviations are possible, but also require brackets. Within specification expressions in the scope of the annotation, one can refer to the defined expression as ::config_OK(conf) for an lvalue conf of correct type. In any case, the latter expression is logically equivalent to its expansion, in which all free occurrences of parameter c are replaced by the actual argument conf.

Symbolic Preprocessor Constants The final kind of annotation is concerned with symbolic preprocessor constants and becomes relevant because of the way we deal with preprocessor macros in CSI and the verification environment. We generally make no assumptions and do not set forth any restrictions about the use of preprocessor macros in program code, except for the provisions of the MISRA-C guidelines [109]. Yet we demand that macros are not referred to inside specifications and that the preprocessor does not modify the latter. This approach works well except in one case, which is due to the fact that C does not have a strict concept of named constants. Variables can be declared with the **const** type qualifier, but that only makes the objects thus defined read-only, and does not allow this 'constant variable' to be used in what is called *constant expressions* [82, § 6.6]. The latter, however, must in particular be used to specify array dimensions. To obtain symbolic names for array dimensions and other numeric values at least in the unprocessed source code, programmers usually use preprocessor macros:

```
#define V_LEN (4 * 1024)
int v[V_LEN];
```

Now, unfortunately, the value of V_LEN will also be of interest inside specifications, e. g. in a memory validity assertion like \array(v2, V_LEN). It generally turned out that one needs access to macros defined as constant numerical expressions within specification expressions. Moreover, during verification one would like to keep the symbolic names and not work with their expansions, both for reasons of readability and because large arithmetic expressions over constants sometimes 'distract' Isabelle's simplification procedures. However, as an artifact of our concrete approach to parsing the source code —where the C preprocessor[3] is run over the code before the actual parsing is done—, we cannot even get access to the symbolic name V_LEN in the code, but only see its expansion.

Our two solutions to this dilemma are shown in Fig. 3.9 and Fig. 3.10. The solution that works with all C preprocessors is to explicitly distinguish between a translation mode for compilation and one for verification. The latter mode is identified by the definedness of a macro VERIFY_MODE. Because during verification the restriction to constant expressions for array dimensions does not apply, we can simply replace macro definitions for numerical constants by regular definitions of type _Int or _Real. To indicate their special character, they are placed inside a special annotation @define. This way, what in compilation mode are macros, are plain variables within specification expressions in verification mode.

Another slightly more elegant solution can be achieved when the preprocessor at hand allows to pass comments inside macro definitions through to expansions and does not silently drop them. In this case one can extend the grammar of C expressions by an annotation postfix operator which indicates that its operand actually arises from a macro expansion. Fig. 3.10 illustrates this: macro LEN will be expanded to

```
(4 * 1024) /*@define _Int LEN = 4 * 1024; */
```

where the annotation represents the postfix operator. Such an expression will then be interpreted as the symbolic integer constant LEN by the parser of the verification environment. As an additional side-effect, such an expression introduces the definition of LEN, so that occurrences of LEN in specification expressions can be associated with the corresponding symbolic constant. It does not raise any problems that macro LEN might be expanded at multiple places, because all (@define) definitions thus introduced are identical and can be merged into one.

[3] usually GNU cpp, available from http://gcc.gnu.org/ at the time of writing.

Chapter 4

Formalised Memory Model for C

In this chapter the memory model is presented which has been developed as the foundation for the formalisation of the semantics of C programs and their specifications. After a brief evaluation of different possible memory models the concept of memory as finite typed maps is described. These are mappings from abstract locations to pairs of representations of C types and sequences of primitive values. Read and write operations on memory are defined and their properties are captured in terms of an update theory given by a collection of conditional rewrite rules. The concept of validity of pointers is introduced. State updates in C evoke the problem of aliasing, i. e. the presence of different programming language expressions referring to a single memory location. We describe how our model allows us to automatically derive the absence of certain kinds of aliasing. Finally, we describe the interplay between representation functions and the memory model.

4.1 Evaluation of Possible Representations

Memory models for imperative languages, which are also called state space models or representations, abound: A simple and intuitive approach is to regard memory as a function from variable names to values, $\mathit{Var} \to \mathit{Value}$, with an appropriate definition of values, e. g. as the union of integers and booleans $\mathbb{Z} \uplus \mathbb{B}$. Such a *functional state model* is fairly generic: by varying either the domain or the range of the function type, the characteristics of the state as relevant to the kind of analysis at hand can be emphasised. When memory can be allocated during program execution, or when variables are first class objects whose address can be computed in programs (as via & in C), the domain typically gets substituted by a set of memory addresses or, more abstractly, *locations* *Loc*, which are also added to the value type. We thus obtain a state model $\mathit{Loc} \to \mathit{Value}$ with $\mathit{Loc} \subseteq \mathit{Value}$. Abstract models of this kind are often used when properties of the programming language itself, e. g. their denotational or operational semantics, are studied. In static program analysis techniques like abstract interpretation, one is often interested in finite, or at least finitely representable *state abstractions* that allow the effective computation of certain properties of concrete programs [114]. Idealised infinite value domains[1] like $\mathbb{Z}$ would be replaced by bounded ones like $I = [int_{min}, int_{max}]$. For example, an interval analysis keeping track of the minimum and maximum values that certain variables can take at particular program points might use a state model in which variable/program point pairs are mapped to an interval $(l, h) \in I \times I$. Even simpler value domains like $\mathcal{P}(\{-, 0, +\})$ might be used for a sign analysis, in which the goal might be to show that certain variable always take on positive values. Here, the value domain is a power set, indicating how uncertainty or imprecision can be incorporated into the functional state model. Due to their approximative character, these simple state models are not appropriate for use in a formal functional verification environment, because they severely restrict the kinds of properties that can be verified.

C is a programming language that provides only a weak abstraction of memory and gives the programmer freedom in breaking even that abstraction: for example, by traversing an aggregate value through a **char** pointer. This leads to another direction in which a memory model can be refined. If C programs shall be analysed for properties pertaining to bit-shifting operations, address arithmetic, pointer conversions, padding bytes of structure types or other low-level details, one is virtually forced to view memory simply as a (finite) sequence of bytes (most commonly 8-bit vectors). This is, e. g., the case in systems code verification, where operating system routines like memory management functions (malloc(), free(), etc.) shall be verified. Viewed functionally, sequences of bytes are mappings $\mathbb{N} \to [0, 255]$. The concretisation of the function domain from abstract locations *Loc* to numbers allows one to express machine-specific details of address arithmetic. For example, if the local variables **int** x, y have the addresses 16 and 28, respectively, and **sizeof**(**int**) == 4 (a 32-bit architecture), then it is possible to derive that the assignment (&x)[3] = 0 will modify y. (Note that according to the standard this expression leads to undefined behaviour; however, systems code targeting a given architecture and compiler might actually include such nasty fragments.) While it is certainly possible to formally model memory and operations thereupon this way, it is rather hard to formally reason about programs at this level of detail. (This holds in particular for the case of interactive verification, where the verifier actually gets to see and work with the memory model.) Tuch [144] formalised a memory model in Isabelle/HOL whose type is $word32 \Rightarrow word8$ at its core, i. e. the sequence-of-bytes model with the additional restriction that there exist only finitely many addresses. He goes to great lengths providing an abstraction layer on top of this model. This layer allows one to have a structured view on memory in terms of access and update functions, that resembles C's aggregate types and that reflects these types in the type system of the theorem prover. So, for example, while a **struct** s { **int** v; **double** x; } is in memory represented as a sequence of bytes, access functions for v and x would be derived that internally take care of computing the relevant offsets into the byte sequence. The amount of work, in particular the accuracy of the formalisation is quite impressive, allowing one even to model padding bytes in structures and the byte order of multi-byte integers. But even though a case study involving a memory allocator has been performed, it seems that the overhead implied by the degree of detail will turn

[1] Note the slightly overlapping terminology here: the *value domain* is in fact the *range* of the functional state.

out to be prohibitive when more abstract functional properties of non-systems code are considered.

We have argued that conceptually it is most natural to model memory as a function, even if the domain (*Var*, *Loc*, *word32*) is finite. For the formalisation of a memory model inside a theorem prover the choice of representation also depends on the facilities that the theorem prover provides with respect to reasoning and simplification over the particular structure. When the domain is finite, e. g., consisting of n variable names, it might be beneficial to use n-tuples, where each variable's value is stored at its corresponding position in the tuple. In Isabelle there is a better alternative by using records, which are essentially tuples with named fields. Read and update functions are derived automatically for newly defined record types, which can then be exploited for the semantics of assignments and expression evaluation. The following record R with fields *varX* and *varY* is equipped with simplification rules such that the subsequent lemma involving an update on field *varX* can be proven automatically:

```
record R = varX :: int
           varY :: bool

lemma "(| varX = 7, varY = True |) (| varX := 8 |) =
       (| varX = 8, varY = True |)"
```

Schirmer [136] developed a verification environment for imperative programs that is independent of a concrete memory model. For practical verification as done in the Verisoft project [5], this generic environment is instantiated with a model based on Isabelle's records. There is a single record for the overall program state. Every global and local variable occurring in a program is represented by a field in this record. For heap-allocated objects, the split heap model of Burstall and Bornat [29] is used: slightly simplifying, the heap is a collection of functions f_1 to f_n from locations to values, one for each structure member f_i occurring in a program. This model is based on the observation that in well-behaved programs, it can never be the case that &s.f == &t.g for different structure members f and g. By introducing a (Isabelle) record field for each such member, the inequality is exploited in the formalisation and the necessary simplification rules come for free, as they coincide with those provided by the record package, as shown by the above example lemma. The disadvantage of this approach is that all objects modelled by record fields, particularly local variables and structure members, are no first class entities: it is not possible to take the address of these objects, because they are constants in the logic of Isabelle/HOL about which no meta-reasoning can be performed. It is therefore not possible to model the way in which a call-by-reference scheme is implemented in C with the record representation, because the way this is typically done is exactly by taking and passing the addresses of objects. If we disallow expressions like &s.f, then we cannot simulate passing `s.f` by reference. The alternatives are either to pass it by value, i. e. to create a copy of `s.f` —which might be expensive if `s.f` contains a large structure—, or to put large objects that are passed between functions into global variables. The latter style leads to brittle and hard-to-read code.

Before we present our memory model, we identify four characteristic properties that this model was supposed to capture. They are

1. to comply with the requirements of the ISO C standard [81] to enable its use in a formal model of a subset of C,
2. to allow to take addresses of lvalues, including local variables &v, structure members &s.f, and array elements &a[i],
3. to represent arbitrarily nested structures and arrays, e. g. **struct** { **struct** { **int** a; } x; **int** a[10]; },
4. to avoid the formalisation and especially reasoning overhead induced by the low-level sequence-of-bytes model in which even simple scalar values like integers can be further destructured.

4.2 Finite Typed Maps

Our state model is *functional*: states are functions with a finite domain (we also call these finite maps), mapping identifiers of *top-level objects* to the representations of their types as well as their values. We distinguish top-level objects from arbitrary objects as defined by the standard as a *"region of data storage in the execution environment, the contents of which can represent values"*. Top-level objects are those defined by local and global variable declarations as well as function parameters. We call them top-level, because they are not contained inside any other object. This is in contrast to, e. g., the objects denoted by structure members, which are contained in the object denoted by the structure itself.

4.2.1 Representing Types

We decided to incorporate type information in the memory model for a similar reason as given by Cohen et al. [44]. In their words, *"in every untyped program, there is a typed program trying to get out"*: While it is possible in C to take a byte-level view of memory, every top-level object has a type given by the corresponding variable definition, that defines its structure. Most of the time —and particularly in code that is not system-related— the program uses its objects in accordance with these defined types. Recording this type information in the program state allows us to determine whether the interpretation of memory locations under particular types is valid or not. For example, we can check against the (disallowed) interpretation of an integer memory location as a floating-point value. Most importantly, we can give a semantics to the CSI predicates \valid and \array of Sec. 3.4.3, i. e. we can define the meaning of pointer and array validity. In addition, we can check for the validity of particular explicit conversions (casts).[2] Imagine a program locally breaks the strict type discipline and converts a pointer of type **int** ∗ to type **void** ∗, i. e. essentially makes it an untyped pointer. At some point this pointer might be converted back, under the assumption that it still points to an object of type **int**, or an array of such objects. The validity of this conversion w. r. t. the types defined by the top-level objects can be checked in the model by looking at the memory type information.

[2] It would even allow us to model an extension of C that provides run-time type information to programs, á la Java's `instanceof` operator; we did not pursue this direction, however.

The C standard distinguishes between *scalar types* and *aggregate types*. The former comprise the arithmetic types (cf. Fig. 3.4) and the pointer types, while the latter are made up of array and structure types. This distinction is reflected in our model: we distinguish between basic types and (run-time) types.[3] The basic types correspond to the scalar types and are used for atomic values (discussed below). They cannot be destructured any further. There are three basic types: *integer*, *floating*, and *pointer* types. The latter are parameterised by the types pointed to. A type, then, is either a basic type, an array type with a given number of elements, or a structure type with a tag name and a list of named members, or *fields*, equipped with types themselves.

```
datatype BasicRTT =
    BR_Int
  | BR_Double
  | BR_Ptr RTT
  and RTT =
    RTT_bas BasicRTT
  | RTT_arr nat RTT
  | RTT_rec string "(string * RTT) list"
```

This is a deep embedding of C types into Isabelle/HOL: the *RTT* datatype provides an explicit tree representation of all C types relevant to our modelling of C values. C types are thus not reflected in the type system of Isabelle/HOL, where there are only the two types *RTT* and *BasicRTT*. We omit types for unions, bit-fields, and functions, since we do not allow them as values in our language subset (cf. Sec. 5.1).

4.2.2 Atomic Values

The distinction between scalar and aggregate types is mirrored in the modelling of values. Scalar values are defined by the datatype *Val*, which constitutes the disjoint union of all integer, floating-point and address (or reference) values:

```
types
  DomInt = int
  DomDouble = real

datatype Val =
    IntVal DomInt
  | DoubleVal DomDouble
  | PtrVal Loc
```

The type definitions for *DomInt* and *DomDouble* identify integer and floating values with Isabelle/HOL's types of mathematical, unbounded integers and real numbers, respectively. Pointer values are addresses, called *locations* in our model. They are defined by the datatype *Loc*, whose definition is given further below. Note that due to the deep embedding of C types the type of locations is not parameterised over the type of values pointed to. The value of every pointer is simply of type *Loc*. All three value types are *atomic* in our model: there is no explicit 'internal representation' of *DomInt* as a sequence of bytes, nor are there conversion functions from byte sequences to integers, or any other value type.

[3] We will use the terms *scalar* and *basic* interchangeably henceforth.

The idealisation of finite machine values as mathematical numbers is arguably a standard approach in interactive formal verification (e. g., [19, 136, 102]). Pragmatically, one can say that this loss in modelling precision (and therefore soundness, an issue we elaborate on in Sec. 5.3) is a necessary price to pay to keep the effort required for the formal verification of systems of realistic size in tolerable bounds. We are not aware of any successful verifications of complex functional properties of realistic programs in a value model that respects integer overflow as well as rounding errors, NaNs and infinities of floating types. To improve on this situation is out of the scope of this thesis. In any case, we do not propose to try and solve all verification tasks within a single tool. It appears much more promising to use separate, specialised tools to obtain partial correctness results, which can be re-used by other tools, a point also stressed in [126]. If, for example, the absence of integer overflow in a concrete program has been shown by classical means, or even proven by a static analysis tool like the one described by Blanchet et al. [25], then the idealisation to mathematical integers in our formal model is no longer unsound for the given program.

The embedding of all scalar values in a single value type *Val* allows for a uniform treatment in the model: all C expressions can be given a semantics so as to yield a *Val*, and memory can be uniformly composed of atomic values, instead of being split into separate areas for separate value types. Concrete contexts will, however, necessitate the interpretation of values as a particular constituent type, e. g. in the semantics of the addition of two integers. We provide operations for the *aggressive evaluation* (a term borrowed from [137]) for all constituent types; under-specified functions allow the projection from *Val* to the respective types:

```
fun
  valToInt :: "Val ⇒ DomInt"
where
  "valToInt (IntVal i) = i"
```

As Isabelle/HOL is a logic of total functions, this function is also logically defined for values not constructed via *IntVal*, but an application such as *valToInt* (*PtrVal p*) cannot be simplified, making the result of the application effectively arbitrary. Under-specification is a common approach to dealing with partiality in logics of total functions [111]. The boolean interpretation of values, on the other hand, is total and complies with the C standard:

```
fun
  is_true_val :: "Val ⇒ bool"
where
  "is_true_val (IntVal n) = (n ≠ 0)"
| "is_true_val (DoubleVal d) = (d ≠ 0)"
| "is_true_val (RefVal r) = not_null r"
```

4.2.3 Flattening of Aggregate Values

The *RTT* datatype suggests a tree-like representation of aggregate values, i. e. objects described by C structure or array types, that mimics the shape of that datatype. The state would then be a function from top-level object identifiers to pairs *RTT* × *StructuredVal*, or even just *StructuredVal*, since the corresponding *RTT* could be directly inferred from the structure of the value. While conceptually this is a nice representation, it turned out that such values are rather unwieldy. Locations (i. e., addresses) of structure members would become paths into trees, such that address arithmetic would become path manipulation. To reflect updates on aggregate values, an update theory for trees would have to be derived. Especially the intricacies of the latter, like deciding what it means to update a tree w. r. t. an invalid path, or simplifying two consecutive updates on overlapping parts of the tree, motivated a simpler representation.

Similar to a concrete machine representation, where aggregate values eventually become sequences of bytes, we represent them as finite sequences of atomic values (*Val*). Every type (RTT) uniquely determines how each aggregate value of that type is represented as a sequence. Intuitively, scalar values are represented by sequences of length one, while the members of structures and arrays are first turned into sequences, which are then concatenated to form the sequence for the aggregate. We can define a function over RTT to formalise the notion of a *flattening* on the type level:

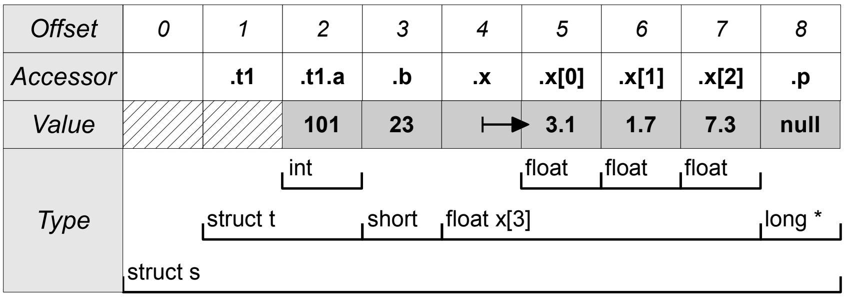
```
fun  flatten_rtt :: "RTT ⇒ RTT list"
and  flatten_frtt :: "(string * RTT) list ⇒ RTT list"
where
  "flatten_rtt (RTT_bas b) = [RTT_bas b]"
| "flatten_rtt (RTT_arr n etype) =
  (RTT_arr n etype # concat (replicate n (flatten_rtt etype)))"
| "flatten_rtt (RTT_rec r fields) =
  (RTT_rec r fields # (flatten_frtt fields))"
| "flatten_frtt ((s, t) # z) = flatten_rtt t @ flatten_frtt z"
| "flatten_frtt [] = []"
```

Function *flatten-rtt* yields a sequence of types, the elements of which define the types of the atomic values in all sequences representing values of the flattened type. This is best described through an example, which is depicted in Fig. 4.1: given the C declarations

```
struct s {
  struct t { int a; } t1;
  short b;
  float x[3];
  long *p;
};

struct s s1 = { { 101 }, 23, { 3.1, 1.7, 7.3 }, NULL };
```

we obtain a sequence of atomic values of length 9 for the aggregate value of `s1`. The type for offset 0 is that of the overall structure, and no atomic value (more precisely: a padding value) is present at offset 0. The insertion of such padding items into sequences makes it possible to assign a *unique type* to every offset of a sequence, a property that will be useful in the derivation of a memory update theory (cf. Sec. 4.3). Following are the sequences for **struct** `s`'s members. The order of the members is that given by the declaration. Hence `.t1` (a **struct** `t`) comes first, which adds another padding value, plus the value of its `.a` member (101). No padding is added at the end of value sequences. Next comes the `.b` member, a **short** with value 23. Note that this atomic value occupies the same amount of 'space' in the sequence as all values of scalar types do. An additional 'padding' value is inserted for arrays, too; in contrast to structures, it is assigned



Offset	0	1	2	3	4	5	6	7	8
Accessor		.t1	.t1.a	.b	.x	.x[0]	.x[1]	.x[2]	.p
Value	▨	▨	101	23	⟼	3.1	1.7	7.3	null

Figure 4.1: Offsets, values, paddings and types for flattened aggregate values

a meaningful value and becomes the *array head pointer*, i. e. the address of the first array element. This enables a smooth modelling of C's *pointer decay*, where evaluating an expression of array type results in the address of the first element. The subsequent offsets 5 to 7 contain the array member values of type **float**. The value of s1 is completed by the null pointer in the .p member.

Our modelling is in slight conflict with the C standard, which requires [82, § 6.7.2.1(13)] that *"a pointer to a structure object, suitably converted, points to its initial member, and vice versa. There may be unnamed padding within a structure object, but not at its beginning."* The benefits of our concrete flattening w. r. t. proving the absence of aliasing (Sec. 4.3) outweighed a strict standard conformance in this case. The implications are that we do not support the mentioned conversion from structure pointers to pointers to their initial member.

Apart from the structure of value sequences, the size of a type is a relevant parameter. It is needed for determining the extent of value representations, and to give a semantics to the **sizeof** operator. A direct definition of the size of a type is given by the following function, which is related to *flatten-rtt* in the obvious manner, as captured by the subsequent lemma:

```
fun sizeof_rtt :: "RTT ⇒ nat"
and sizeof_rtt_fields :: "(string * RTT) list ⇒ nat"
where
  "sizeof_rtt (RTT_bas brtt) = 1"
| "sizeof_rtt (RTT_arr n rtt) = 1 + n * sizeof_rtt rtt"
| "sizeof_rtt (RTT_rec name fields) = 1 + sizeof_rtt_fields fields"
| "sizeof_rtt_fields [] = 0"
| "sizeof_rtt_fields ((_, rtt)#z) = sizeof_rtt rtt + sizeof_rtt_fields z"
```

```
lemma flatten_sizeof_rtt: "length (flatten_rtt ty) = sizeof_rtt ty"
```

This lemma is proven by induction over *ty*. Specialised induction rules for type *RTT* following the recursive structure of the definition of *flatten-rtt* are derived by Isabelle/HOL automatically. Many proofs pertaining to structural properties of the memory model, specifically to the possible aliasing between locations, do not require induction, but can be based on a non-recursive property which we call the *preorder property.* A list *rs* of *RTT* has the preorder property, if for all partitionings of *rs* into an initial segment *xs*, a single item *t*, and the remainder *ys* it is the case that the flattening of *t* is an initial segment of the

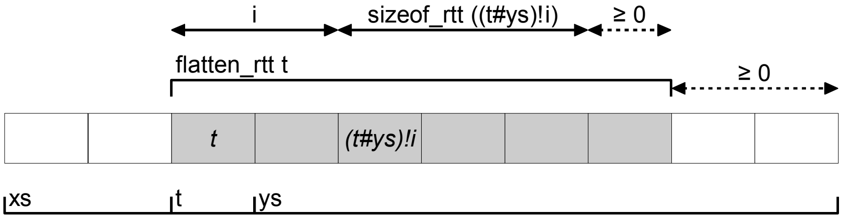


Figure 4.2: The preorder property: all elements t are followed by their flattenings, bounded in size. The flattenings of elements contained in t's flattening are themselves contained in the latter.

list $t\#ys$. Formally:

```
definition
  preorder_prop :: "RTT list ⇒ bool"
where
 "preorder_prop rs =
   (∀xs t ys. rs = xs @ (t#ys) ⟶
     (sizeof_rtt t ≤ length (t#ys) ∧
        (take (sizeof_rtt t) (t#ys)) = flatten_rtt t))"
```

Importantly, we can prove that type flattenings have the preorder property:

```
theorem flatten_rtt_preorder_prop:
  "preorder_prop (flatten_rtt rtt)"
```

Given the partitioning into xs, t, ys from above, we can therefore show that the size of all types $(t\#ys)\,!\,i$ found in the flattening of t is bounded by the difference between the size of t and the respective position i. This formalises the intuition of a recursive containment property: in the flattening of a (type) tree, we find the flattenings of all descendants of the root node of the tree, which themselves contain the flattenings of all of their descendants. This property is depicted in Fig. 4.2 and formalised by the following lemma. It allows us to prove the inequality between all locations denoted by nested members of a structure, like `s1.t1.a` and `s1.b`.

```
lemma preorder_prop_bounded:
  "preorder_prop rs ⟹
    (∀xs t ys. rs = xs @ (t#ys) ⟶
      (∀i < sizeof_rtt t. (sizeof_rtt ((t#ys)!i) + i ≤ sizeof_rtt t)))"
```

4.2.4 Locations

The final ingredient of the memory model is the modelling of addresses, or locations. A common (mis-)conception about pointer arithmetic in C is that more or less any memory location can be reached from any other location by an appropriate pointer addition or subtraction. One must however distinguish between standard conforming behaviour and things that ‘work’ in concrete implementa-

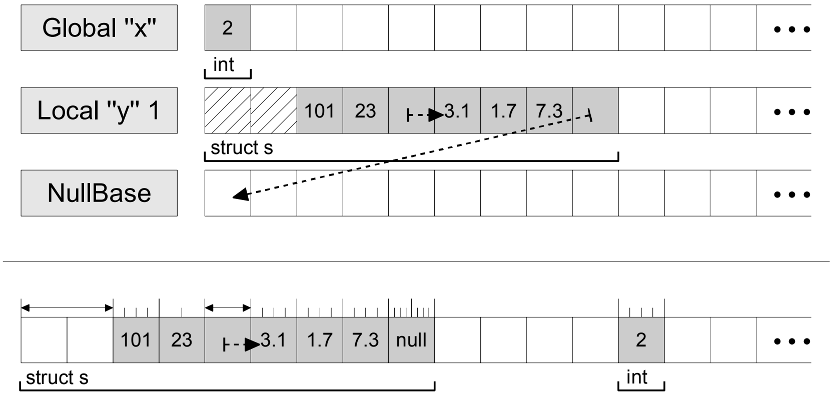


Figure 4.3: Distinct top-level objects are unrelated in our memory model (top); one cannot reach *Local y* through pointer arithmetic on, e. g., *Global x*. A memory-as-array model (bottom), on the other hand, would enable this.

tions. The C standard actually only defines pointer arithmetic [82, § 6.3.2.3, § 6.5.6] for two cases: (1) the addition of an integer value and a pointer as well as the subtraction of an integer or a pointer from another pointer are defined if the pointers point to members of the same array object[4], and the result remains in the boundaries of the array (or exceeds them by one) (2) a pointer to any object can be converted to a pointer to character type, which can be used to traverse all bytes of the object.

Both cases do not enforce the use of a fully linear memory-as-array model. While we must ignore (2) due to the idealisation of numbers, (1) only requires the 'arithmetic reachability' between objects within the same array. Hard-wiring separation information into the address model aids verification, because the possibilities of aliasing are reduced. We therefore model addresses in two steps: every top-level object has a unique *base location*, identified by the variable's name (and a number for local variables which can have equal names in distinct functions). A special base location is assigned to the null pointer, making the treatment of null pointers conceptually simple. No special null-cases need to be considered when computing offsets on locations. This is sound, since pointer arithmetic on the null pointer is not defined. A *location* consists of a base location and an offset into the sequence representing atomic and aggregate values. Fig. 4.3 visualises the address model and contrasts it to the memory-as-array model.

```
datatype BaseLoc = Global string
                 | Local string nat
                 | NullBase

datatype Loc = Loc BaseLoc nat
```

[4] Single objects are identified with arrays of size one in this case.

Global and local variables can be constructed via helper functions that also introduce convenient syntax: global variables are prefixed with a γ, while locals are identified by a leading ν. Pointer arithmetic is only performed on a single base location: it is not possible to reach another base location by adding offsets. *base-loc* and *loc-offset* are selector functions on locations.

```
definition globalVar :: "string ⇒ Loc"  ("γ·_" [199] 200)
where "globalVar idt = Loc (Global idt) 0"

definition localVar :: "string ⇒ nat ⇒ Loc"  ("ν·__" [0,0] 200)
where "localVar idt n = Loc (Local idt n) 0"

fun
  add_offset :: "Loc ⇒ nat ⇒ Loc"
  base_loc :: "Loc ⇒ BaseLoc"
  loc_offset :: "Loc ⇒ nat"
```

Array access and field selection

We need operations on locations that correspond to the access of the members of structures and arrays to encapsulate the actual offset computations involved in these accesses. We provide an *array access* and a *field selection* operation for these purposes. An array access on locations, written $m_t.[i]$, should not be confused with a C array access m[i] (for m an array with element type t): the former does not involve a dereferencing on location m, but directly computes the offset for member i on m itself. This allows it to be a state-independent operation. m must therefore already be the location of an array element; it must not be the location of the overall array, due to the array head pointer. Field selection expects the location of a structure and yields the location of the given field through an offset computation. Both operations are parameterised over a type, which is required for the offset computations.

```
definition
  array_acc :: "Loc ⇒ RTT ⇒ nat ⇒ Loc"
                ("__.[_]" [120,120,0] 119)
where
  "array_acc loc rtt idx = add_offset loc (sizeof_rtt rtt * idx)"

definition field_sel :: "Loc ⇒ RTT ⇒ string ⇒ Loc"
            ("__→_" [120,200,120] 119)
where "field_sel l rtt fld =
  (case rtt of
    (RTT_rec name fields) ⇒
      add_offset l (field_offset fields fld 0 + 1)
   | _ ⇒ l)"
```

Array access and field selection provide a view on memory that is close to that of C lvalues, except for the lack of a state-dependent dereferencing operation. Proof obligations for programs that adhere to the type structure given by the declarations of top-level objects will only mention location terms built via these operations, as in $((l_t \to f)_u \to g)_v.[i]$; no explicit offset computations will be visible.

4.2.5 Memory as Finite Maps

Finally, we can introduce the type of program states, i. e., our memory model.

```
typedef State =
  "{f :: BaseLoc ⇀ (RTT × (nat ⇀ Val)). finite (dom f)}"
```

The memory part of a program state is a partial function with a finite domain, mapping base locations to their type and a partial function from offsets to atomic values. The latter represents the flattening sequence of aggregate values and will practically also have a finite domain. By using partial functions (instead of lists) we can easily model the 'dummy items' by excluding their offset from the function's domain.[5] We do not distinguish between stack-allocated and heap-allocated objects, since we treat the addresses of both as first-class citizens and therefore aim for a uniform treatment.

The basic operations on memory are reading and writing as well as allocating and deallocating memory areas. Our strategy is to derive a simple theory of the memory operations (which we call the (memory) *update theory*) and to defer all sanity checks like the definedness of updated locations or type constraints to the program semantics (cf. Sec. 5.3). While we obviously want to disallow invalid memory accesses, we do not handle them at the low level of the update theory. Practically, this means that all memory operations are defined as total functions, where the effect of 'invalid' operations is defined so as to ease the update theory. This leads to the following signatures of the basic memory functions.

```
definition
  read :: "Loc ⇒ State ⇒ Val"
  update :: "Loc ⇒ Val ⇒ State ⇒ State"
  extend :: "Loc ⇒ RTT ⇒ State ⇒ State"
  dealloc :: "Loc ⇒ State ⇒ State"
```

Memory updates are modelled functionally, i. e. updates transform input states to modified output states. Memory extension is defined at the level of base locations, i. e. we can only extend the state by a full base location and a corresponding type. (The function signature expects a location only for more convenient usage.) It is not possible to 'enlarge' memory at an existing base location. Deallocation means un-defining the memory function at a given base location. We can always obtain a *fresh location* which is not yet mapped in a given memory, because memory functions always have a finite domain. We do not model restrictions in the memory size of real machines. Freshness plays an important role in solving aliasing constraints; for example, the local variables of a function will be fresh for the state in which the function was called, and hence cannot be aliased to lvalues denoting 'older' objects.

```
definition
  is_fresh_loc :: "Loc ⇒ State ⇒ bool"  (infix "∉_S" 120)
where
  "l ∉_S σ = (base_loc l ∉ dom (Rep_State σ) ∧ (∃ x m. l = ν·x_m))"

lemma
 is_fresh_loc : "(fresh_loc x σ) ∉_S σ"
```

[5] Recall that partial functions are modelled in Isabelle/HOL as total functions into the *option* datatype, which is syntactically hidden within the ⇀ arrow.

Fig. 4.4 contains the basic update theory, consisting of a series of conditional rewrite rules. It makes use of the infix syntax for memory operations, which are $l\,@_i\,\sigma$ for *read-int* $l\ \sigma$ (accordingly $l\,@_d\,\sigma$ is written for *read-double* $l\ \sigma$ and $l\,@_l\,\sigma$ for *read-loc* $l\ \sigma$), $\sigma(l ::= v)$ for *update* $l\ v\ \sigma$, $\sigma \oplus (l, t)$ for *extend* $l\ t\ \sigma$, and $\sigma \ominus l$ for *dealloc* $l\ \sigma$. Reading integer values is defined in terms of *read* by applying aggressive evaluation. The theory contains the expected theorems: reading a location just updated with a value yields that value; reading a different location allows us to 'step over' an update; the same holds for extensions. Stepping over a deallocation requires an inequality of base locations, since deallocation sweeps a whole top-level object away. Updates on distinct locations commute (*update-commute*), while a second update on a single location cancels the first one. Deallocation and update as well as extension on locations with distinct bases commutes, while deallocation cancels updates and extensions on the same base.

4.2.6 Valid Pointers and Arrays

The concept of validity of pointers and arrays is required for two reasons: firstly, pointers of a given type may only be dereferenced if an object of the respective type is found at the location pointed to. A similar condition holds for valid array accesses, where additionally the array index needs to be taken into account. Secondly, with regard to aliasing we want to derive inequalities between lvalues based on their syntactic structure. In particular, we desire the *split heap property* [29], stating that memory can be partitioned by the field names of structures, so that `a.f` and `b.g` can never be aliased for `f` ≠ `g`. This does not hold unconditionally, so that `a` and `b` again have to be restricted to be valid locations.

The validity of a location r is only defined w. r. t. a type t and a state σ. To be *valid in* σ, r must not be a null pointer and t must be equal to the type at r in σ as determined by the type flattening:

```
definition
  valid_loc :: "Loc ⇒ RTT ⇒ State ⇒ bool"
               ("•’(_ _|_’)" [200,200] 200)
where
  "valid_loc r t σ = (not_null r ∧ t ∈ (state_rtt_at σ r))"
```

where operation *state-rtt-at* $\sigma\ r$ yields a singleton set consisting of the type in the flattening at r, or the empty set if r describes an undefined location in σ.

The concept of a valid location is extended to valid pointers in the obvious way: a pointer value *PtrVal* p is valid if its target p is a valid location. The definition of a valid array access is a little more intricate: we cannot expect to always deal with complete arrays in C (as would be the case in, e. g., Java), but pointers may contain the addresses of arbitrary array elements and be used to access further elements of the array. The following listing illustrates this:

```
int v[10];
int *p = &v[3];
p[2] = 7;
```

```
definition
  read_int :: "Loc ⇒ State ⇒ DomInt"
where
 "read_int l σ = valToInt (read l σ)"


lemma read_int_update:
  "l @i (σ(l ::= (IntVal v))) = v"


lemma read_int_update_other:
  "l ≠ m ⟹ l @i (σ(m ::= v)) = l @i σ"


lemma read_int_extend_other:
  "l ≠ l1 ⟹ l @i (σ⊕(l1, r)) = l @i σ"


lemma read_int_dealloc_other:
  "base_loc l1 ≠ base_loc l2 ⟹ l1 @i (σ⊖l2) = l1 @i σ"


lemma update_commute:
  "l ≠ la ⟹  (σ(la ::= va, l ::= v)) = (σ(l ::= v, la ::= va))"


lemma update_cancel:
  "((σ(l ::= v'))(l ::= v)) = (σ(l ::= v))"


lemma dealloc_update_other:
  "base_loc l ≠ base_loc l1 ⟹ (σ(l1 ::= v))⊖l = (σ⊖l)(l1 ::= v)"


lemma dealloc_update_cancel:
  "base_loc l  = base_loc l1  ⟹ (σ(l1 ::= v))⊖l = σ⊖l"


lemma dealloc_extend_cancel:
  "l ∉S σ ⟹ (σ⊕(l, v))⊖l = σ"


lemma dealloc_extend_other:
  "⟦l1 ∉S s; base_loc l ≠ base_loc l1⟧ ⟹
    ((σ⊕(l1, v))⊖l) = ((σ⊖l)⊕(l1, v))"
```

Figure 4.4: The update theory for *DomInt*; according theories also exist for *DomDouble* and *Loc*.

Program safety requires both that `p[2]` is a valid integer location, but also that the array index 2 applied to `p` does not exceed the overall size of `v`. The latter condition is necessary, since if `v` was nested in some larger structure whose flattening contains subsequent valid integer locations, a too large index could leave the array `v` and still end up at such a location. Though seemingly well-behaved in our particular memory model, this is generally undefined behaviour and must be ruled out. We therefore define the validity of an array access w. r. t. the location of an array element r, the array element type t, a state σ and the index i. That r actually is an array element can be expressed as the fact that it can be reached from a location l denoting an array through an array access (with index k); index i is valid if $k+i$ does not exceed the array length as given by the type at l. Formally:

definition
```
  valid_array_acc :: "Loc ⇒ RTT ⇒ int ⇒ State ⇒ bool"
                     ("•[_ _|_|_]" [0,0,0,0] 200)
```
where
```
"valid_array_acc r t i σ =
    (∃l k len. not_null r ∧ 0 ≤ i ∧ r = (add_offset l 1)_t.[k]
        ∧ (RTT_arr len t) ∈ state_rtt_at σ l
        ∧ k + (nat i) < len)"
```

The validity of an array access implies location validity:

lemma *valid_array_acc_valid_loc:*
```
  "⟦ •[r_t| n| σ]; 0 ≤ i; i ≤ n ⟧ ⟹ •((r_t.[nat i])_t| σ)"
```

State-dependent functions require theorems about their interplay with state modifications. Validity is not affected by updates and extensions on the state, so for example we have:

lemma *valid_loc_update:*
```
  "•(r_t| σ) ⟹ •(r_t| (σ(l ::= v)))"
```

On the other hand, deallocation may obviously destroy validity; it is only preserved for locations with distinct base locations. Validity can be introduced by extending a state by a fresh location. This location will then be valid for the type by which the state was extended together with the location.

lemma *valid_loc_addressof:*
```
  "l ∉_S σ ⟹ •(l_t| (σ⊕(l, t)))"
```

Together with *valid-loc-update* this yields the validity of function-local variables in all program states during function execution.

4.3 Location Inequalities

An update theory like that of Fig. 4.4 allows us to calculate the value stored in memory at a given location, under the condition that we can decide the (in)equality of locations. We start with a couple of seemingly trivial, immediately decidable inequalities, each concrete instance of which yet requires a tiny proof via the lemmas given below:

1. Two local or two global variables (or field selections and array accesses thereupon) are distinct due to their different names: $x \neq y \Longrightarrow \nu{\cdot}x_t \neq \nu{\cdot}y_u$
2. Local and global variables are distinct: $\nu\cdot x_t \neq \gamma\cdot y$

3. A fresh and a valid location are unequal and have distinct base locations:

   $[\![\ \bullet(r_{t}|\ \sigma);\ l \notin_{S} \sigma]\!] \Longrightarrow \mathtt{base\_loc}\ r \neq \mathtt{base\_loc}\ l$

4. Two locations that are valid for distinct types are not equal:

   $[\![\ \bullet(r1_{t1}|\ \sigma);\ \bullet(r2_{t2}|\ \sigma);\ t1 \neq t2\ ]\!] \Longrightarrow r1 \neq r2$

   This is a rather notable property of our memory model, and a direct consequence of the definition of location validity, which only allows the unique type given by the respective type flattening to be the valid type at a location. It follows that the above lemma is only applicable in the presence of pointers that obey the type structure determined by the top-level objects, which is the most common case.

4.3.1 Structures and Arrays

The fact that our definition of a type flattening has the preorder property allows us to prove the property that is the foundation of the split heap approach. While our model does not enforce the inequality between field selections of differently named fields by construction, this property is well derivable[6]:

theorem *valid_loc_field_name_neq:*
$"[\![\ \bullet(r1_{RTT\_rec\ n1\ flds1}|\ \sigma);$
$\quad \bullet(r2_{RTT\_rec\ n2\ flds2}|\ \sigma);$
$\quad f1 \in \mathtt{fst}\ `\ \mathtt{set}\ flds1;\ f2 \in \mathtt{fst}\ `\ \mathtt{set}\ flds2;\ f1 \neq f2]\!]$
$\quad\ \Longrightarrow r1_{RTT\_rec\ n1\ flds1}\!\rightarrow\! f1 \neq r2_{RTT\_rec\ n2\ flds2}\!\rightarrow\! f2"$

The inequality of field selection locations can be reduced to the inequality of field names, at least if the field selections are done on valid locations. The additional restriction that $f1$ and $f2$ must be fields that occur in the respective types is due to our deep embedding of types and field names which formally allows a selection of field f on a structure type that does not have this field. Such side-conditions are proven automatically for all relevant types and are stored as auxiliary theorems by the verification environment.

We can also prove that a field selection and an array access location are unequal, given they descend from a corresponding valid location and array:

theorem *field_sel_neq_array_acc:*
assumes va: $"\bullet[l_{ta}|\ n|\ S]"$ **and**
$\qquad vr$: $"\bullet(r_{RTT\_rec\ nm\ fs}\ |\ S)"$ **and**
$\qquad ffs$: $"f \in \mathtt{fst}\ `\ \mathtt{set}\ fs"$ **shows**
$"r_{RTT\_rec\ nm\ fs}\!\rightarrow\! f \neq l_{ta}.[\mathtt{nat}\ n]"$

For inequalities between two array access locations the situation is not so clear cut. Two valid array accesses into arrays of distinct types can be shown to be unequal via theorem *valid-array-acc-valid-loc* and inequality 4. (distinct pointer types) of the previous section. For arrays of the same element type, however, we essentially rely on an explicit separation assertion (cf. Sec. 3.4.2).

If we try to derive an inequality between to field selections E.f and D.g via the above theorem, we need to be able to derive that both E and D evaluate to valid locations. In the most common cases, this information is either available

[6] The proof of this *split heap property* is by no means immediate from the definitions; nevertheless, we do not consider the intermediate lemmas relevant in the context of this thesis, and hence omit them.

because E and D are some local variables a and b, which are always valid thanks to *valid-ref-addressof*, or because they are pointers whose validity is ensured by the function precondition. (In the latter case, the corresponding C expressions would actually be (∗a).f, resp. a−>f, and for b accordingly.) If, however, E is itself a field selection c.h on some variable c, we end up having to show the former is a valid location. We therefore need to be able to derive the validity of locations obtained via field selections from the validity of the location of the whole structure. This nesting property holds in our memory model, as demonstrated by the following theorem:

```
lemma valid_loc_field_sel_nested:
  "⟦ •(l_{RTT_rec nm fs}| σ); field_type fs f = Some t ⟧ ⟹
     •((l_{RTT_rec nm fs}→f)_t| σ)"
```

A similar point also applies to nested array accesses, and theorems to derive the validity of an array access from the validity of the overall array have been shown. The following demonstrates a variant which states that &l[k] is a valid array reference of size n for the 'tail' of the overall array l of size k + n:

```
lemma valid_array_acc_of_array_acc:
  "⟦•[l_t| int k + n| σ]; 0 ≤ n ⟧ ⟹ •[l_t.[k]_t| n| σ]"
```

4.4 Non-Atomic State Modification

So far, modifications on the state were atomic in the sense that they only affected single locations in the case of updates and single base locations in the case of state extensions and deallocations. But while every finite set of non-atomic modifications could be encoded as a sequence of atomic ones, it turns out to be more convenient to reason about non-atomic modifications directly. Such *modification sets* naturally occur in a modular verification methodology in the shape of effect summaries, of which modifies clauses are an example: the effect of a function call on the program state is bounded by the function's @modifies clause. Hence, every function call (just like every loop invariant or @join predicate) introduces a modification set during verification. Their main use is not in precisely specifying the changes that exist between memories, but rather complementary to capture the common part of program memories.

This is formalised in two parts, relating to the values at locations and the types, respectively. Two states are invariant w. r. t. a given modification set X :: *Loc set* if the functional restrictions of their value parts (the second projection of the overall state function) to the complement of X are equal. Expressed extensionally, this means that both state functions yield equal values (or are both undefined) for all locations not in X.

```
definition
  inv_state_eq :: "Loc set ⇒ State ⇒ State ⇒ bool"
where
  "inv_state_eq X S S' =
     (state_read S ↾ (-X) = state_read S' ↾ (-X))"
```

For types, which determine the structure of a state, we are not only interested in the equality of type mappings outside a given modification set X, but we also want to ensure that the structure on all base locations occurring in X

is compatible. The idea is to create a relation in which two states are only related if one can be transformed into the other through a sequence of updates and extensions, because these are exactly the states we need to relate during program verification. This notion of compatibility can be expressed as follows: S and T are structurally compatible over X if the types given by S are equal to those given by T on all base locations outside those occurring in X, and if the type mapping given by S, restricted to the base locations in X, is a sub-mapping (denoted by the symbol $\subseteq_m$) of that of T. Intuitively, this merely forces the type mapping of S to be the same as that of T, except for a possibly smaller definedness domain w. r. t. base locations occurring in X.

```
definition
  struct_state_le :: "Loc set ⇒ State ⇒ State ⇒ bool"
where
  "struct_state_le X S T =
    (let B = base_loc ‘ X in
      state_typing S ↾ (- B) = state_typing T ↾ (- B) ∧
      state_typing S ↾ B ⊆_m state_typing T ↾ B)"
```

The actual relation we are concerned with is simply the conjunction of invariance and structural compatibility. This defines a partial order:

```
definition
 modified_on :: "Loc set ⇒ State ⇒ State ⇒ bool"
where
"modified_on X S T = (struct_state_le X S T ∧ inv_state_eq X S T)"
```

```
lemma modified_on_refl: "T ⊑_X T"
```

```
lemma modified_on_trans :
  "⟦ S ⊑_X T; T ⊑_X R ⟧ ⟹ S ⊑_X R"
```

The relevance of this concept of a partial ordering on states, parameterised over a modification set, lies in the fact that during verification all intermediate states of the symbolic execution of a function are related by this ordering. We obtain a generalisation of lemmas *read-int-update* and *valid-loc-update*, which allow us to ignore a single update when reading a location or proving its validity, for states related via a whole modification set:

```
lemma mod_read_int_eq:
 "⟦ S ⊑_X T; l ∉ X ⟧ ⟹ l @i T = l @i S"
```

```
lemma mod_valid_loc:
"⟦ S ⊑_X T; •(l_t/ S) ⟧ ⟹ •(l_t/ T)"
```

Obviously, concrete states will be related via different modification sets, which makes the above transitivity rule considering just one set inapplicable in practice. Fortunately, the partial ordering is monotone w. r. t. the modification set, which allows for a more algorithmic transitivity rule which relates states via the union of two modification sets:

```
lemma modified_on_mono:
  "⟦ X ⊆ Y; S ⊑_X T ⟧ ⟹ S ⊑_Y T"
```

```
lemma modified_on_trans_Un:
  assumes A1: "σ ⊑_X σ’"
  and A2: "σ’ ⊑_Y σ’’"
```

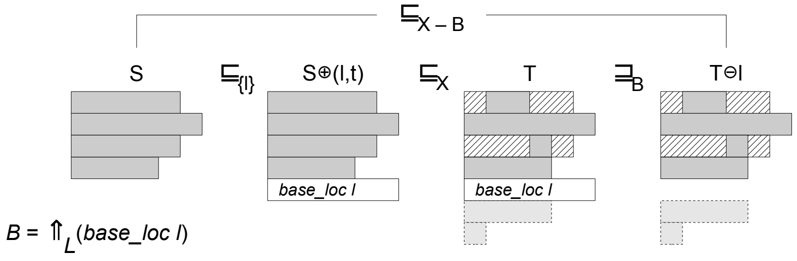


Figure 4.5: Relation between a state S before the allocation of l, and a state $T\ominus l$ resulting from the extension $S\oplus(l,t)$, further updates and extensions on X, and the deallocation of l.

shows "$\sigma \sqsubseteq_{(X\ \cup\ Y)} \sigma''$"

Lemmas *mod-read-int-eq* and *modified-on-trans-Un* suggest a (heuristic) algorithmic procedure to determine the value at a particular location l in a state T: find a $\sqsubseteq_X$-path, i. e., a sequence of states $S \sqsubseteq_X S_1 \sqsubseteq_X S_2 \sqsubseteq_X \cdots \sqsubseteq_X S_n \sqsubseteq_X T$, such that the value of l in S is known and that $l \notin X$. This procedure needs to make the implicit relation between a state and its atomically updated, extended, and deallocated successors explicit, which looks as follows:

lemma *update_modified_on:*
"$\sigma \sqsubseteq_{\{l\}} (\sigma(l\ ::=\ v))$"

lemma *extend_modified_on:*
"$\sigma \sqsubseteq_{\{l\}} (\sigma\oplus(l,t))$"

lemma *dealloc_modified_on:*
"$X = \Uparrow_L(\mathit{base\_loc}\ l) \Longrightarrow \sigma\ominus l \sqsubseteq_X \sigma$"

where operation $\Uparrow_L$ yields all locations that can be built from a given base location. Note that updated and extended states are larger than their corresponding initial states, while a deallocated state is smaller. The typical execution of a function will first allocate some local variables, perform a series of updates both atomic and non-atomic (e. g., via function calls), and then deallocate the local variables, leading to a sequence of intermediate states σ_0, $\sigma_1 = \sigma_0\oplus(l_1, t_1)$, $\sigma_2 = \sigma_1\oplus(l_2, t_2)$, …, $\sigma_k = \sigma_{k-1}(l_k ::= v)$, $\sigma_k \sqsubseteq_{X_1} \sigma_{k+1}$, …, $\sigma_{n-1} = \sigma_{n-2}\ominus l_2$, $\sigma_n = \sigma_{n-1}\ominus l_1$. To rewrite $l_0 \,@_i\, \sigma_n$ to $l_0 \,@_i\, \sigma_0$, with l_0 not in the union of all modification sets induced by this sequence, we require a further lemma. It must allow us to relate the two states that exist before a particular location has been allocated and after that location has been deallocated, like σ_0 and σ_n, or σ_1 and σ_{n-1}. Fig. 4.5 illustrates the situation: modified locations X are depicted by hatchings; newly allocated locations are coloured in a lighter gray; S and $T\ominus l$ are structurally the same on all (base) locations on which S is defined. The according lemma is as follows:

lemma *modified_on_extend_dealloc:*
"$[\![\ S\oplus(l,t) \sqsubseteq_X T;\ l \notin_S S\ ]\!] \Longrightarrow S \sqsubseteq_{X\ -\ \Uparrow_L(\mathit{base\_loc}\ l)} T\ominus l$"

4.5 Representation Functions and Memory

The update theory presented so far provides a useful basis for determining the (atomic) values at particular locations in states that occur during program verification. It does, however, not consider the relation between non-atomic values created through representation functions and memory updates. Consider a representation function *Vec2DR* reading the x- and y-coordinates of a C structure of type *Vec2D-t* and yielding their values as a complex number *Complex x y*: we want to be able to automatically derive that the complex number read in a state where both coordinates have been updated consists of exactly the updated values, i. e.,

$$\mathit{Vec2DR}\ (\sigma(l_{\mathit{Vec2D\text{-}t}}{\to}x ::= v,\ l_{\mathit{Vec2D\text{-}t}}{\to}y ::= u))\ l = \mathit{Complex}\ v\ u$$

This is an example where there exists a close correspondence between the concrete C representation and its interpretation as a domain value. Another example is the interpretation of arrays as collections, for which iteration and indexing are defined. Each representation function over such a data type can be equipped with an elegant update theory, which we will present below.

Abstract data types

We must contrast these structurally straightforward representation functions with those that work over abstract data types, like binary heaps, balanced trees, or graphs. For the latter arbitrary updates of the concrete C data structure are often not well-defined within the domain abstraction. To illustrate the problem we consider the abstract data type of a binary heap (of integer values), implemented as a structure containing the number of elements currently in the heap and an array representing a complete binary tree:

```
typedef struct heap {
  int size;
  int val[MAX];
} heap_t;
```

A typical heap property would state that in a heap `h` the child nodes of all nodes in the tree (reachable for node `i` at indices `2 * i` and `2 * i + 1`) have larger values than their parents

```
\forall int i; 0 <= i && i < h.size ==>
  (2 * i < h.size ==> h.val[i] <= h.val[2 * i]) &&
  (2 * i + 1 < h.size ==> h.val[i] <= h.val[2 * i + 1])
```

We have to distinguish between code that uses heaps versus the code implementing the heap library. In both cases, an update theory considering arbitrary updates on the `heap_t` structure will not be useful. Client code will only perform specific kinds of updates that are encapsulated in high-level operations (e. g., insertion, deletion) maintaining the above *data type invariant.* During verification of the client code, one will only make use of the specifications of these high-level operations, which abstract from particular updates and rather describe the effect on the respective domain value. Specific update rules for the data structure are only necessary for the verification of the library, i. e., the high-level operations themselves. Every abstract data type requires its own such set of specific rules, making a uniform treatment virtually impossible. As the code verified in the SAMS project mainly used data types with a rather close correspondence between representations and their domain abstractions, we concentrated on these.

Partial updates

We now turn back to the cases where a direct correspondence between C aggregate types and their domain interpretation exists. The general structure of the required simplification rules for representation functions follows that of the update theory for atomic values. Some rules state when an update can be ignored, i. e., when $R\ \sigma(l' ::= v)\ l$ equals $R\ \sigma\ l$ for a representation function R. We call all locations for which this equation does not hold the *dependence set* of the representation function. Other rules exist for the cases where the representation is affected by an update. At this point we face a problem regarding partial updates of representations: if R depends on locations $l_1, \ldots, l_n$ $(n > 1)$, and one of them gets updated, we obtain a term $R\ \sigma(l_i ::= v_i)\ l$. We would now like to express this update in terms of the value that R yields for the non-updated state. Effectively, we are asking for a collection of update functions f_{l_i}, one for each location l_i in the dependence set of R, that satisfy

$$R\ \sigma(l_i ::= v_i)\ l = f_{l_i}\ (R\ \sigma\ l)\ v_i$$

This structure would allow a continuation of the update rewriting process on $R\ \sigma\ l$, where we assume $\sigma = \sigma'(l_2 ::= v_2)$, finally yielding a term

$$f_{l_1}\ (f_{l_2}\ (\cdots (R\ \sigma_0\ l) \cdots)\ v_2)\ v_1$$

in which either $R\ \sigma_0\ l$ has become irrelevant for the overall value, since it is determined completely by the updates f_{l_i}, or where the necessary properties about $R\ \sigma_0\ l$ can be derived, e. g., from a given precondition. Reading such a value would proceed along an update theory over the f_{l_i}. Typically however, domain types do not provide such update functions for their components; this holds true in particular for newly defined types. Even if update functions are available, the theories over these functions will generally vary slightly, which makes them impractical for use in an automatic update simplification procedure.

We solved this problem for a class of representation functions predominantly used in programs that do not perform dynamic memory allocation. Functions in this class have a finite dependence set, where all locations in this set can be expressed as (sequences of) accesses on a single location, which we call the function's *handle*. Examples of representation functions that belong in this class are the above *Vec2DR* and generally all those that build a domain value from a single C structure or a single array. Essentially we rule out pointer-connected structures of dynamic size, like linked lists or trees.

4.5.1 Representations as Structures

The solution to the problem of obtaining a coherent theory for partial updates on representation functions is to build these in a two-step process. The first step is state-dependent and transforms the set of values at the locations in the dependence set into an intermediate representation in the shape of an Isabelle-/HOL record. In the second step the record representation is simply lifted into the actual domain type. Every representation function $R :: \mathit{State} \Rightarrow \mathit{Loc} \Rightarrow \alpha$ can thus be written as a composition $R\ \sigma\ l = \tau\ (\varrho\ \sigma\ l)$. Accordingly, we will speak of ϱ-functions and τ-functions in the following. Note that the range of every ϱ-function as well as the domain of every τ-function is a record type. Since records do have a uniform update theory, we obtain a systematic correspondence between state updates that partially modify the representations of domain values and updates on their intermediate record representation.

We walk through the necessary lemmas along the example of a representation function for 2D vectors. At the moment these lemmas need to be proven by hand for each concrete type, but their structure allows for a completely automatic derivation. First, we define the intermediate record and a function *Vec2DRec* (written $\varrho(\mathit{vec})$ in the following) creating records given a state and a handle:

```
record Vec2DRec =
  Xcoord :: real
  Ycoord :: real

definition
"Vec2DRec σ l =
   (let x = (lVec2D_t→''x'') @d σ;
        y = (lVec2D_t→''y'') @d σ
    in
      ⦇Xcoord = x, Ycoord = y⦈)"
```

The following update lemmas formalise the equivalence between a state update on a location in the dependence set and a corresponding record update:

```
lemma Vec2DRec_update_x:
 "ϱ(vec) (σ(lVec2D_t→''x'' ::= DoubleVal v)) l =
    ϱ(vec) σ l ⦇Xcoord := v⦈"
lemma Vec2DRec_update_y:
 "ϱ(vec) (σ(lVec2D_t→''y'' ::= DoubleVal v)) l =
    ϱ(vec) σ l ⦇Ycoord := v⦈"
```

Updating locations outside the dependence set does not affect the value yielded by a representation function. The following lemma suggests a point-wise inequality proof between the updated location and all locations in the dependence set; sometimes it may be more efficient to prove the inequality between base locations, for which according lemmas exist, too.

```
lemma Vec2DRec_update_other:
 "l' ∉ {lVec2D_t→''x'', lVec2D_t→''y''} ⟹
  ϱ(vec) (σ(l' ::= v)) l = ϱ(vec) σ l"
```

A disjointness condition arises for the proof of equality between two domain values read in states related via $\sqsubseteq_X$:

```
lemma Vec2DRec_modified_on:
 "⟦σ' ⊑X σ; {lVec2D_t→''x'', lVec2D_t→''y''} ∩ X = {}⟧ ⟹
  ϱ(vec) σ l = ϱ(vec) σ' l"
```

The effect of extensions and deallocations on representation functions is a direct consequence of lemmas *{extend,dealloc}-modified-on* and *Vec2DRec-modified-on* above.

```
lemma Vec2DRec_extend_other':
 "l' ∉ {lVec2D_t→''x'', lVec2D_t→''y''} ⟹
```

```
  ϱ(vec) (σ⊕(l',t)) l = ϱ(vec) σ l"

lemma Vec2DRec_dealloc_other:
 "base_loc l ≠ base_loc l' ⟹ ϱ(vec) (σ⊖l') l = ϱ(vec) σ l"
```

The presented rules allow us to perform update simplification on representation function terms. Each step introduces inequality and disjointness side-conditions. The discussion about how these are solved in a concrete verification is deferred until Sec. 6.5.2. Nevertheless, it can already be seen that all side-conditions can be reduced to inequalities w. r. t. location terms built up through access operations on the handle of the representation function. We conclude the presentation of the update theory for representation functions that work on a single structure with the definition of the τ-function $\tau(vec)$ that lifts *Vec2DRec* records to complex numbers, and the final composition of $\varrho(vec)$ and $\tau(vec)$ to form the actual representation function *Vec2DR*, which can be referred to in CSI specifications.

```
definition Vec2DRecToComplex :: "Vec2DRec ⇒ complex"
                                ("τ'(vec')")
where
  "τ(vec) r = (Complex (Xcoord r) (Ycoord r))"

definition Vec2DR :: "State ⇒ Loc ⇒ complex"
where
  "Vec2DR σ l = (τ(vec) (ϱ(vec) σ l))"
```

4.5.2 Representations as Arrays

The situation is not quite as straightforward when representation functions read arrays. We concentrated on the case where arrays are used to hold *collections* of elements, and distinguish three cases: first, the *order* of elements in an array can be relevant or not; moreover, the presence of *duplicate elements* may play a role for unordered collections, a case we silently assume when dealing with ordered collections. In all three cases lists can be used as intermediaries, introducing the problem of reflecting partial updates on representations within lists.

First we define how a list is generated from the elements of an array. There is a generic variant that is parameterised over the representation function for the array members, but here we concentrate on the specific incarnation that reads lists of intermediaries via $\varrho(vec)$. No τ-function is applied to the list members yet, but instead a τ-function for the whole list of intermediaries will be provided. This is important: an intermediary shall be a value for which an update theory exists (i. e., it shall consist of records and lists); after applying a τ-function this property is lost. Reading a list of n elements works thus:

```
fun
Vec2DRec_n :: "State ⇒ Loc ⇒ nat ⇒ Vec2DRec list"
              ("ϱ'(vecs')")
where
"ϱ(vecs) σ l 0 = []" |
"ϱ(vecs) σ l (Suc n) =
  (ϱ(vec) σ l # ϱ(vecs) σ (l_Vec2D_t.[1]) n)"
```

The transformation of such a list of records into a domain value depends on the intended interpretation of the array. In the simplest case, we are actually interested in lists of complex numbers. In that case, the τ-function is just a mapping of $\tau(vec)$ onto all list elements (see definition below). Another important interpretation is that of a set of complex numbers, in which case this mapping would be followed by an application of function $set :: 'a\ list \Rightarrow 'a\ set$, which is predefined in Isabelle/HOL.[7]

definition
```
Vec2DRList :: "State ⇒ Loc ⇒ DomInt ⇒ complex list"
```
where
```
"Vec2DRList σ l n = map τ(vec) (ϱ(vecs) σ l (nat n))"
```

By induction over n we can prove the obvious fact about the length of the intermediate list, as well as the expected fact that at each index i of the list we find the $\varrho(vec)$ intermediary of the ith array member.

lemma `Vec2DRec_n_length:`
```
  "length (ϱ(vecs) σ l n) = n"
```

lemma `Vec2DRec_n_nth:`
```
  "i < n ⟹ ϱ(vecs) σ l n =
        ϱ(vecs) σ l i @
        ϱ(vec) σ (lVec2D_t·[i]) # ϱ(vecs) σ (lVec2D_t·[i+1]) (n-(i+1))"
```

To prove that an intermediate list built with $\varrho(vecs)$ is independent of an atomic update, we must show that the updated location is not in the dependence set of any array element:

lemma `Vec2DRec_n_update_other:`
```
  "⟦ ∀i < n. la ∉ {(lVec2D_t·[i])Vec2D_t→''x'',
                     (lVec2D_t·[i])Vec2D_t→''y''} ⟧ ⟹
     ϱ(vecs) (σ(la ::= v)) l n = ϱ(vecs) σ l n"
```

This formulation yields a side-condition that is easy to prove in two steps if it is known that la and l lie in distinct base locations, or if la is known to be the selection of a field other than x or y, i. e., $la = l'_u \rightarrow f$ $(f \notin \{"x", "y"\})$, thanks to lemma *valid-loc-field-name-neq*.

An update on field x (or y) of element i in the array leads to an update of the ith value in the intermediate list. We can therefore delegate the update to the ρ-function of that value, and update the list at index i with its result.

lemma `Vec2DRec_n_update_x_i:`
```
  "⟦ i < n; la = (lVec2D_t·[i])Vec2D_t→''x'' ⟧ ⟹
  ϱ(vecs) (σ(la ::= v)) l n =
   (ϱ(vecs) σ l n)[i := ϱ(vec) (σ(la ::= v)) (lVec2D_t·[i])]"
```

Lists of complex numbers are built by nested applications of ϱ-functions, so that the delegated update itself can be simplified by an application of update rule *Vec2DRec-update-x*, leading to the following rewrite rule that completely transformed the state update into an update of the ith list element by a record with an updated *Xcoord*.

lemma `Vec2DRec_n_update_x_i':`
```
  "i < n ⟹
  ϱ(vecs) (σ((lVec2D_t·[i])Vec2D_t→''x'' ::= DoubleVal v)) l n =
   (ϱ(vecs) σ l n)[i := ϱ(vec) σ (lVec2D_t·[i]) (| Xcoord := v |)]"
```

[7] Note that *Vec2DRList* is a representation function that is used in specification expressions, which explains why its argument type is *DomInt* instead of *nat*.

Analogous to atomic updates that do not affect the value of a representation function, there is a rule dealing with states related via $\sqsubseteq_X$, which follows the structure of *Vec2DRec-modified-on*, but now obviously requires the disjointness between X and a dependence set of an arbitrary array element:

```
lemma Vec2DRec_n_modified_on:
  "⟦ σ ⊑X σ';
    ∀i < n. X ∩ {(lVec2D_t·[i])Vec2D_t→''x'',
                  (lVec2D_t·[i])Vec2D_t→''y''} = {} ⟧ ⟹
    ϱ(vecs) σ' l n = ϱ(vecs) σ l n"
```

At this point a concrete example is in order to demonstrate the use of the update simplification rules presented so far, and also to illuminate the remaining limitations. First, we look at a procedure that swaps two vectors in an array. The precondition ensures that i and j are valid indices for the array vs, and that i != j, while the postcondition demands that the elements are actually swapped — this is such a low-level property that it can hardly be expressed in any other way than using the list update operations directly (recall that ^^R{..} refers to an evaluation of representation function R in the pre-state):

```
/*@
  @requires \array(vs, len)
    && $max(i, j) < len && i != j
  @modifies vs[i], vs[j]
  @ensures ${ ^Vec2DRList{vs, len} =
    ^^Vec2DRList{vs, len}[i := ^^Vec2DR{vs[j]},
                          j := ^^Vec2DR{vs[i]}] }
  @*/
void vec_swap(Vec2D_t *vs, int i, int j) {
  double tmp;
  tmp = vs[i].x; vs[i].x = vs[j].x; vs[j].x = tmp;
  tmp = vs[i].y; vs[i].y = vs[j].y; vs[j].y = tmp;
}
```

Taking v as the location pointed to by vs and writing t for Vec2D_t for brevity, then during the verification of this procedure we will obtain a sequence of states that directly reflects the assignments in terms of state updates:

$$\begin{aligned}
&\sigma_0 \quad \textit{(initial state)}\\
&\sigma_1 = \sigma_0(\mathit{tmp} ::= v_t.[i]_t{\rightarrow}x \mathbin{@_d} \sigma_0)\\
&\sigma_2 = \sigma_1(v_t.[i]_t{\rightarrow}x ::= v_t.[j]_t{\rightarrow}x \mathbin{@_d} \sigma_0)\\
&\sigma_3 = \sigma_2(v_t.[j]_t{\rightarrow}x ::= \mathit{tmp} \mathbin{@_d} \sigma_0)\\
&\sigma_4 = \sigma_3(\mathit{tmp} ::= v_t.[i]_t{\rightarrow}y \mathbin{@_d} \sigma_0)\\
&\sigma_5 = \sigma_4(v_t.[i]_t{\rightarrow}y ::= v_t.[j]_t{\rightarrow}y \mathbin{@_d} \sigma_0)\\
&\sigma_6 = \sigma_5(v_t.[j]_t{\rightarrow}y ::= \mathit{tmp} \mathbin{@_d} \sigma_0)
\end{aligned}$$

Here we assume for simplicity that all read operations have been rewritten to read the initial state (e. g., $\mathit{tmp} \mathbin{@_d} \sigma_i = \mathit{tmp} \mathbin{@_d} \sigma_0$) and that all arising inequality side-conditions (e. g., $\mathit{tmp} \neq v_t.[i]_t{\rightarrow}x$) have been proven. The specified postcondition gets expanded to the following equation, referring to the pre-state σ_0

and the post-state σ_6:

$$\begin{aligned}&\mathit{Vec2DRList}\,\sigma_6\ v\ (\mathit{nat}\,n) =\\ &\quad \mathit{Vec2DRList}\,\sigma_0\ v\ (\mathit{nat}\,n)\ [\,i := \mathit{Vec2DR}\ \sigma_0\ v_t.[j], j := \mathit{Vec2DR}\ \sigma_0\ v_t.[i]\,])\end{aligned}$$

The equation is proven by applying the update simplification rules introduced above to the term on the left-hand side. We obtain

$$\begin{aligned}
&\mathit{Vec2DRList}\,\sigma_6\ v\ n\\
\overset{(1)}{=}\ &\mathit{map}\ \tau_{vec}\ (\varrho_{vecs}\ \sigma_6\ v\ (\mathit{nat}\,n))\\
\overset{(2)}{=}\ &\mathit{map}\ \tau_{vec}\ (\varrho_{vecs}\ \sigma_5\ v\ (\mathit{nat}\,n)\\
&\qquad\qquad [j := (\varrho_{vec}\ \sigma_5\ v_t.[j])(\!|\,\mathit{Ycoord} := v_t.[i]_t{\to}y\ @_d\,\sigma_0|\!)])\\
\overset{(3)}{=}\ &\mathit{map}\ \tau_{vec}\ (\varrho_{vecs}\ \sigma_4\ v\ (\mathit{nat}\,n)\\
&\qquad\qquad [i := \varrho_{vec}\ \sigma_4\ v_t.[i](\!|\,\mathit{Ycoord} := v_t.[j]_t{\to}y\ @_d\,\sigma_0|\!),\\
&\qquad\qquad\ j := \varrho_{vec}\ \sigma_5\ v_t.[j](\!|\,\mathit{Ycoord} := v_t.[i]_t{\to}y\ @_d\,\sigma_0|\!)])\\
\overset{(4)}{=}\ &\mathit{map}\ \tau_{vec}\ (\varrho_{vecs}\ \sigma_2\ v\ (\mathit{nat}\,n)\\
&\qquad\qquad [j := \varrho_{vec}\ \sigma_2\ v_t.[j](\!|\mathit{Xcoord} := \mathit{tmp}\ @_d\,\sigma_0|\!),\\
&\qquad\qquad\ i := \varrho_{vec}\ \sigma_4\ v_t.[i](\!|\,\mathit{Ycoord} := v_t.[j]_t{\to}y\ @_d\,\sigma_0|\!),\\
&\qquad\qquad\ j := \varrho_{vec}\ \sigma_5\ v_t.[j](\!|\,\mathit{Ycoord} := v_t.[i]_t{\to}y\ @_d\,\sigma_0|\!)])\\
\overset{(5)}{=}\ &\mathit{map}\ \tau_{vec}\ (\varrho_{vecs}\ \sigma_0\ v\ (\mathit{nat}\,n)\\
&\qquad\qquad [i := \varrho_{vec}\ \sigma_4\ v_t.[i](\!|\,\mathit{Ycoord} := v_t.[j]_t{\to}y\ @_d\,\sigma_0|\!),\\
&\qquad\qquad\ j := \varrho_{vec}\ \sigma_5\ v_t.[j](\!|\,\mathit{Ycoord} := v_t.[i]_t{\to}y\ @_d\,\sigma_0|\!)])
\end{aligned}$$

where step (1) is the unfolding of *Vec2DRList*, steps (2) and (3) are applications of *Vec2DRec-n-update-x-i'*. In step (4) we have skipped the update on *tmp* in σ_4 and applied *Vec2DRec-n-update-x-i'* again. We can see that this earlier list update at index j will not have an effect on the overall list value, since it will be overwritten by the latter update at the same index. The same argument holds for the update at index i induced by the update in σ_2. Step (5) reflects this; no further simplification of the term $\varrho_{vecs}\ \sigma_0\ v\ (\mathit{nat}\,n)$ is possible. We continue with the simplification of $\varrho_{vec}\ \sigma_4\ v_t.[i]$ by applying the rules for records of Sec. 4.5.1.

$$\begin{aligned}
\varrho_{vec}\ \sigma_4\ v_t.[i] &= \varrho_{vec}\ \sigma_3\ v_t.[i]\\
&= \varrho_{vec}\ \sigma_1\ v_t.[i]\ \ (\!|\mathit{Xcoord} := v_t.[j]_t{\to}x\ @_d\,\sigma_0|\!)\\
&= \varrho_{vec}\ \sigma_0\ v_t.[i]\ \ (\!|\mathit{Xcoord} := v_t.[j]_t{\to}x\ @_d\,\sigma_0|\!)
\end{aligned}$$

Analogously, we can rewrite

$$\varrho_{vec}\ \sigma_5\ v_t.[j] = \varrho_{vec}\ \sigma_0\ v_t.[j]\ \ (\!|\mathit{Xcoord} := v_t.[i]_t{\to}x\ @_d\,\sigma_0|\!).$$

We substitute the simplified terms back into the above equation (step (6)) and obtain

$$
\begin{aligned}
& \mathit{map}\ \tau_{vec}\ (\varrho_{vecs}\ \sigma_0\ v\ (\mathit{nat}\ n) \\
& \qquad\qquad [i := \varrho_{vec}\ \sigma_4\ v_t.[i]\llparenthesis\, \mathit{Ycoord} := v_t.[j]_t{\to}y\ @_d\ \sigma_0 \rrparenthesis, \\
& \qquad\qquad\ j := \varrho_{vec}\ \sigma_5\ v_t.[j]\llparenthesis\, \mathit{Ycoord} := v_t.[i]_t{\to}y\ @_d\ \sigma_0 \rrparenthesis]) \\
\overset{(6)}{=}\ & \mathit{map}\ \tau_{vec}\ (\varrho_{vecs}\ \sigma_0\ v\ (\mathit{nat}\ n) \\
& \qquad\qquad [i := \varrho_{vec}\ \sigma_0\ v_t.[i]\llparenthesis \mathit{Xcoord} := v_t.[j]_t{\to}x\ @_d\ \sigma_0\ , \\
& \qquad\qquad\qquad\qquad \mathit{Ycoord} := v_t.[j]_t{\to}y\ @_d\ \sigma_0\ \rrparenthesis, \\
& \qquad\qquad\ j := \varrho_{vec}\ \sigma_0\ v_t.[j]\llparenthesis \mathit{Xcoord} := v_t.[i]_t{\to}x\ @_d\ \sigma_0\ , \\
& \qquad\qquad\qquad\qquad \mathit{Ycoord} := v_t.[i]_t{\to}y\ @_d\ \sigma_0\ \rrparenthesis]) \\
\overset{(7)}{=}\ & (\mathit{map}\ \tau_{vec}\ (\varrho_{vecs}\ \sigma_0\ v\ (\mathit{nat}\ n))) \\
& \qquad\qquad [i := \tau_{vec}\ (\varrho_{vec}\ \sigma_0\ v_t.[i]\llparenthesis \mathit{Xcoord} := v_t.[j]_t{\to}x\ @_d\ \sigma_0\ , \\
& \qquad\qquad\qquad\qquad\qquad \mathit{Ycoord} := v_t.[j]_t{\to}y\ @_d\ \sigma_0\ \rrparenthesis), \\
& \qquad\qquad\ j := \tau_{vec}\ (\varrho_{vec}\ \sigma_0\ v_t.[j]\llparenthesis \mathit{Xcoord} := v_t.[i]_t{\to}x\ @_d\ \sigma_0\ , \\
& \qquad\qquad\qquad\qquad\qquad \mathit{Ycoord} := v_t.[i]_t{\to}y\ @_d\ \sigma_0\ \rrparenthesis)] \\
\overset{(8)}{=}\ & \mathit{Vec2DRList}\ \sigma_0\ v\ (\mathit{nat}\ n)\ [\,i := \mathit{Vec2DR}\ \sigma_0\ v_t.[j], j := \mathit{Vec2DR}\ \sigma_0\ v_t.[i]\,]
\end{aligned}
$$

In step (7) we have applied *map-update*, allowing us to lift τ_{vec} into the list updates:

$$\mathit{map}\ f\ \mathit{xs}[i := v] = (\mathit{map}\ f\ \mathit{xs})[i := f\ v] \qquad (\mathit{map\text{-}update})$$

Step (8) simply folds the definition of *Vec2DRList* and makes use of the fact that the updates on *Xcoord* and *Ycoord* with values read in σ_0 completely determine the records, which coincide with the records yielded by $\mathit{Vec2DR}\ \sigma_0\ v_t.[i]$ and $\mathit{Vec2DR}\ \sigma_0\ v_t.[j]$, respectively. Given the sequence of states σ_0 to σ_6 we can thus prove the postcondition by applying the rules presented in this section. How the states are obtained is the topic of Sec. 6.5

Iterating Over Arrays

We have demonstrated the use of the update theorems for representation functions given updates of specific array elements. The simplification steps that were taken in this example are applied automatically by the verification environment. We now need to consider the more common case of array manipulation within a loop iterating over the array elements. For example, in the SAMS project arrays of 2D vectors were mostly used to represent point sets. Individual points in these sets were hardly ever considered in isolation. By iterating over the corresponding arrays, geometric transformations over such sets could be implemented, as in

```
for (i = start; i < end; ++i) {
  /* Some initial computations */
  v[i].x = fx( ... );
  v[i].y = fy( ... );
}
```

where each individual iteration does not alter the values of array elements with smaller indices than that of the current one. The typical structure of an invariant for such a loop will look as follows

```
/*@
  @invariant start <= i && i <= end &&
   \forall int j; start <= j && j < i —> ::P(v, j)
  @ // ...
  @*/
```

where ::P(v, j) denotes an arbitrary property of the domain value of v[j]. After the loop, one can then conclude that property ::P is true for all array elements between start and end. The critical part is the invariant step, i. e., the proof that the invariant is maintained by a single loop iteration. The lemmas presented so far are not immediately applicable in such a proof; however, they allow the derivation of the following more suitable lemma:

lemma *Vec2DRList_prop_step':*

```
  "⟦ ∀x ∈ set (Vec2DRList σ l (int n)). P x;
     σ ⊑X σ';
     ∀i < n. X ∩ {(l Vec2D_t.[i]) Vec2D_t→''x'',
                  (l Vec2D_t.[i]) Vec2D_t→''y''} = {};
   P (Vec2DR σ' (l Vec2D_t.[n])) ⟧ ⟹
  ∀x ∈ set (Vec2DRList σ' l (int n + 1)). P x"
```

It directly reflects the schema of the invariant step: we want to show P holds for all complex numbers in the set obtained by reading array l up to $n+1$ in the state σ' after the iteration, while knowing that it holds in σ for all elements up to n. σ and σ' are related via X, corresponding to the @modifies clause of the loop. The two conditions to be proven are that X is disjoint from the array elements with lesser indices (meaning that the iteration does not alter previous results, as described above), and that the domain value for array element n actually has property P in σ'.

Lemma *Vec2DRList-prop-step'* follows from *Vec2DRec-n-modified-on* and *Vec2DRec-n-nth*, which emphasises the use of these more basic lemmas, even if they are only of limited use in automatic update simplification tactics, due to the fact that array index computations translate to computations on list indices, so that index inequality is generally not decidable.

This concludes the presentation of the memory model. We only presented a sketch of the most important theorems related to the memory model in this chapter, and skipped all the proofs. We have done so because we do not consider the 'history' of these higher-level properties, i. e. the technical lemmas they depend upon, as interesting to readers of this thesis. However, we emphasise that the formalisation of the memory is complete in the sense that all theorems shown here and used in the actual verification, e. g., in the SAMS project, have been proven is Isabelle/HOL without the introduction of any axioms.

Chapter 5

C Programs and Specifications in Isabelle/HOL

In this chapter the subset of the C programming language that is supported by the verification environment is identified. Furthermore the formalisation of C programs and CSI annotations in Isabelle/HOL is discussed. The formalisation incorporates a monadic denotational semantics of program execution based on state transformers, which allows for side-effects in expressions. Since this is a standard approach, only the more specific parts are shown. Finally, a simpler semantics for side-effect free expressions is developed that is more appropriate for use in the proof rules of the next chapter. A syntactic criterion is defined that ensures the equivalence of the two semantics.

5.1 Language Subset

Most formalisations of the C language effectively define neither a subset nor a superset of the actual language. On the one hand, peculiar language features like bit-fields or variable function arguments are often excluded due to a lack of theoretical interest or practical relevance. Bessey et al. [22] sportfully discuss how even a minor omission of a language feature can render a static analysis tool unusable for commercial application domains. On the other hand, restrictions imposed by the standard out of practical considerations, like the maximum number of function arguments or the length of identifiers, might be irrelevant for the purposes of a formalisation. In any case, a mathematically precise definition of the set of C programs can only be given on the syntactic level, since the semantics of C are only informally defined by the standard. Nevertheless, we desire a formalisation of the program semantics w. r. t. which we can prove the correctness of our program logic that is used for program verification.

Formal Semantics Several formalisations of the semantics of C exist, both on paper and within tools for mechanised logic [117, 26, 136, 68]. They differ in the style of semantics (denotational, big-step or small-step operational) and the number of language features supported. Formalisations aiming at a more or less complete coverage are often concerned with a meta-level analysis of the semantics itself, while others tailor the semantics so as to smoothen its use in subsequent steps, like the derivation of a program logic, or the proof of equivalence w. r. t. another semantics. Important decisions concern the complexity of the memory model, which influences the possible amount of pointer arithmetic and low-level memory operations, and how the non-determinism inherent in the standard is dealt with. For example, the fact that expression evaluation is left unspecified essentially leaves the choices of fixing an evaluation order, which incurs restrictions on the allowed side-effects in expressions, and of dealing with several possible evaluations, hence non-determinism. Acknowledging the frequency with which new formalisations of programming language semantics are developed, Sewell et al. [139] even enable the definition of the formal semantics of a programming language in a general framework, which allows the evaluation rules to be typeset and automatically be translated to the input language of several theorem provers, including Isabelle/HOL and Coq.

Our verification environment is to be used in the certification of C programs and our aim was not to develop a particularly powerful or novel formal semantics for C. Rather, it should be easily explainable to and understood by external reviewers. To this end, it builds on well-understood mathematical principles, and supports exactly those language features that appeared in the programs we verified. We proceed in two steps: first, we discuss prominent features of C and point out whether or in which way they are recognised by the verification environment. Then we provide the definitive reference in terms of the abstract syntax and formalised denotational semantics.

5.1.1 Discussion of Language Features

In the following, we mark those features whose inclusion would cause a substantial extension of the formalisation of the semantics by a $\star$.

Integer overflow and unsigned integers The semantics of integer overflow differs between the signed and unsigned integer types. While a signed expression like INT_MAX + 1 exhibits implementation-defined behaviour, its unsigned counterpart UINT_MAX + 1U has a well-defined wrap-around semantics and yields 0. In accordance with the memory model, our semantics entirely ignores integer overflow, so that the above expressions evaluate to INT_MAX + 1 and UINT_MAX + 1 (both of type *DomInt*), respectively. We have argued in Sec. 4.2.2 that the proof of freeness of overflow is delegated to other tools. The SAMS code did not rely on wrap-around semantics, so that it was natural to prohibit *all* kinds of overflow.

Implicit and explicit arithmetic conversions All arithmetic conversions can remain *implicit* in C. Their behaviour is equal to explicit conversions and is defined as long as the value to be converted is in the range of values representable by the target type (possibly after truncation in the case of a conversion of a floating-point type to an integer type). For example, after the assignment sequence

```
double d = 100.1;
```

```
  char c = d;
```

c is implicitly converted from **double** to **char** and will contain the value 100. Additionally, the arguments of arithmetic operations are subject to *integer promotions*, as explained in Sec. 3.3.3. To avoid the unexpected consequences entailed by such conversions, we *require* all conversions from types with a larger set of representable values to a type with a smaller such set to be made explicit. This property is checked on the syntactic level by the front-end. Further, we *assume* that all conversions are value-preserving and do not result in undefined behaviour. All conversions for which the first assumption does not hold effectively perform a truncation or modulo operation, which can also be achieved by calling appropriate functions.

Bit-level operations The verification environment does not support bit-level operations (>>, <<, ~, &, |, and ^). There are no theoretical objections against their inclusion for unsigned integers. However, their semantics requires a modelling of the bit-sizes of integer types, which is an architecture-dependent property. For example, x >> 40 is undefined if x is an unsigned integer of bit size 32. Furthermore, value preservation is not meaningful when operating on the bit-level, because leading 0s become relevant. Consider the code snippet

```
  unsigned char c = UCHAR_MAX; /* Bits: 1111 1111 */
  unsigned int j1 = ~c;
  unsigned int j2 = (unsigned char)(~c);
```

Only j2 will actually obtain the value 0, while j1 will have all except the last 8 bits set to 1, due to the implicit conversion of c from **unsigned char** to **unsigned int** in the second line.[1]

Unions and bit-fields* Unsurprisingly, C has a low-level concept of union datatypes. They simply allow one to view a single chunk of memory as different types of objects. C unions are untagged —in contrast to discriminated unions of functional languages like OCaml and to variant records of Pascal—, so that the correct access to a union datatype is not enforced at the type level. Given a union

```
union u {
  int x;
  double v;
} u1;
```

the compiler cannot detect in general when u1.v is read even though u1.x was last written to. A serious formal treatment of unions would therefore include the supervision of a correct access scheme for unions. What is worse is that the type of object contained in a union can be changed at run-time by an appropriate assignment. This wreaks havoc on any type-based aliasing analysis: in the context of the above union all pointers to **int** would become possible pointers to **double** values. The inequalities of memory locations of Sec. 4.3 rely on the uniqueness of the type at each location and would therefore become invalid if unions were integrated into the model.

[1] We assume that a byte consists of 8 bits throughout this chapter.

Bit-fields are reportedly seldom used [142, Q. 2.26]; their inclusion is therefore not deemed necessary.

Type qualifiers and storage class specifiers The type qualifier **volatile** is not supported. Static function-local declarations are not supported directly, but such declarations are lifted to file scope by the front-end which also performs an appropriate renaming of the declared identifier. This is a semantically transparent transformation. The storage class specifier **register** is simply ignored, as the memory model does not distinguish registers from other memory.

Pointer conversions* The implementation-defined conversion of a pointer to an integer and back ([82, § 6.3.2.3]) is not supported, just as the conversion from a structure pointer to a pointer to its initial member is not allowed. The latter restriction is necessary in any memory model that has the split heap property: if it were allowed, the following code

```
struct inner { int x; } *ip;
struct outer { struct inner y; } *op;
ip = (struct inner *) op;
```

would introduce the location equality between op−>y and ip−>x, i. e. between two locations obtained via accesses on distinct field names.

Pointer polymorphism C allows the implicit and explicit conversion of any pointer to object type to the type **void** ∗ and back. This is used both to implement type-agnostic memory operations as well as ‘pointer polymorphism’. By the former we mean operations which interpret their **void** ∗ arguments simply as the address of a memory area on which to operate in a low-level manner (like the copy operation memcpy() or memset(), which sets all bytes of a memory area to a given value). Our memory model is not suited for such operations. By pointer polymorphism we mean C’s simple variant of parametric polymorphism [1], i. e., the use of type **void** ∗ in the parameter types of functions whose behaviour does not depend on the actual types of the pointers thus passed. A classic example for the usefulness of this principle is an operation that reverses the elements of an array (which must be of pointer type), which can be generically declared over an array of **void** pointers, avoiding the need for separate definitions for individual types.

```
void reverse_array(void *v[], int len);
```

Similarly, one can introduce parametric datatypes like one for tuples of pointers:

```
struct tuple { void *fst; void *snd; };
```

The verification environment therefore supports the conversion of any object pointer to type **void** ∗ and back.

Functions* In accordance with the MISRA guidelines, we do not allow direct or mutual recursive function definitions. Since function pointers are also excluded, this property can be checked on the syntactic level by the front-end during a whole program analysis. The use of functions with variable parameter numbers is prohibited as well. Structurally, functions may only have a single exit point at the end of the function body. This is also enforced syntactically by requiring that every function definition must end with a unique **return** statement.

Expressions of scalar type While objects of aggregate type may be defined both locally and globally, we demand that all expressions are of *scalar*, i. e. either arithmetic or pointer, types. This entails that aggregate assignments need to be translated by the front-end to a sequence of assignments to all scalar components comprising the object to be modified. Furthermore, functions may only accept parameters and return values of scalar type. Since the address of any aggregate object may be taken, these objects can always be made available across function calls by passing their address.

Evaluation order and side-effects in expressions The evaluation order of the operands of binary expressions and of function arguments is left unspecified by the standard. In combination with side-effects in expressions, this introduces non-determinism into the language. The MISRA guidelines demand that an expression must yield the same value under every possible order of evaluation of its subexpressions, providing the following example where the result is supposed to depend on whether the left-hand or right-hand side of the addition are evaluated first:

```
x = b[i] + i++;
```

We note that this expression yields undefined rather than unspecified behaviour [82, § 6.5(2)], as i is both read and written to within a single expression.[2] In fact, Norrish [118] has shown that all expressions not containing internal sequence points deterministically evaluate to a value, or else expose undefined behaviour.

In program logics it is common practice to restrict all expressions to be completely free of side-effects of any kind, including function calls. This is done to allow the direct use of such expressions inside specifications, as exemplified by the use of condition b in the classic Hoare rule for a simple **if** statement:

$$\frac{\{b \wedge P\}\ c_1\ \{Q\} \quad \{\neg b \wedge P\}\ c_2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q\}}$$

This approach enforces a peculiar programming style using many auxiliary variables, which is not suitable in a context in which many trivially side-effect free mathematical operations are used. Consider the 2D-rotation of an x-coordinate, and its roundabout computation avoiding function calls in expressions:

```
x = cos(phi) * x - sin(phi) * y;
/* vs. */
double t1 = cos(phi);
double t2 = sin(phi);
x = t1 * x - t2 * y;
```

Worse cases are easily conceived. We therefore take a more relaxed approach that still avoids all cases of undefinedness. Concretely, we allow calls of functions with a **@modifies** \nothing specification to appear in expressions, which

[2] Addition for language lawyers: and i is not only read to determine the value to be stored in it.

includes a large class of mathematical operations. The assignment and increment/decrement operators may, on the other hand, only be used as the top-level operators of an expression statement.

Finally, in accordance with the MISRA guidelines, the comma operator is not supported.

Dynamic memory management* We are compliant with the MISRA guidelines by *excluding* dynamic memory management via malloc() and free(). However, the memory model itself provides most necessary operations: *extend* and *dealloc* exactly correspond to allocation and deallocation, while *is-fresh-loc* permits the statement that a given location is not yet defined in a program state. What is missing are a relation between program states (similar to $\sigma \sqsubseteq_X \sigma'$, e. g. $\sigma \rhd_Y \sigma'$) to express which locations Y have been deallocated in a post-state σ' w. r. t. pre-state σ, as well as appropriate theorems about the interplay between this relation and read operations on the program state.

Statements* Non-local exits, embodied by the **break**, **continue** and **return** statements, are not treated in the formalisation. They do not pose theoretical problems, and could be dealt with through the techniques described in [79, 149]. While the abrupt termination of a loop iteration is sometimes convenient for the programmer, the same effect can always be achieved by an appropriate cascade of **if** statements. The code of the SAMS project did not suffer from clumsy workarounds or unreadability by omitting non-local exit statements. Unconditional jumps via **goto** cannot be catered for as easily, but we consider their use as bad practice anyway. The **switch** statement permits arbitrary jumps to any point in the statement body, making it behave more or less like a **goto**. We only support the modified, Java-like syntax for **switch** statements as defined in [109, § 6.15], which is internally translated into a cascade of **if** statements during parsing. The internal translation of **for** and **do** ... **while** loops to their equivalent **while** loop counterparts has been described in Sec. 3.5.2.

C preprocessor The C preprocessor is applied to all sources before they are passed to our front-end. Hence, all preprocessor usage as accepted by the MISRA guidelines is acceptable. The special treatment of arithmetic constants defined as preprocessor macros has already been described in Sec. 3.5.3.

5.2 Abstract Syntax

The abstract syntax of C programs is represented as a collection of Isabelle/HOL datatypes. Roughly, they fall into the four categories of abstract syntax for types, expressions, statements, and declarations. Fig. 5.1 shows types and expressions.

5.2.1 Types and Expressions

We have seen the datatype for types of values in the memory model, *RTT*, already (cf. Sec. 4.2.1). There is another datatype *Type*, which is output by the front-end translating the concrete source code into Isabelle/HOL, which is nearly

```
datatype BasicType = IInt | IDouble | IVoid
datatype Type = BasicType BasicType
              | RefType Type
              | RecordType Identifier "Recordfield list"
              | ArrayType Type nat
    and Recordfield = Recordfield Identifier Type

datatype ArithOp = OpPlus | OpMinus | OpMult | OpDiv | OpMod
datatype CompOp  = Less | LessEq | Eq | NotEq

datatype LVal = LId Identifier Type
              | LArrayAcc RefExpr IntExpr Type
              | LDeref RefExpr Type
              | LFieldSel LVal Identifier Type
  and IntExpr = IntLVal LVal
              | IntLit int
              | IntFunCall Identifier ExprList
              | IntUMinus IntExpr
              | IntArith ArithOp IntExpr IntExpr
              | IntCond Expr IntExpr IntExpr
              | IntIntComp CompOp IntExpr IntExpr
              | IntDoubleComp CompOp DoubleExpr DoubleExpr
              | IntRefEq RefExpr RefExpr
              | IntRefDiff RefExpr RefExpr Type
              | IntLAnd Expr Expr
              | IntLOr Expr Expr
              | IntLNot Expr
              | DoubleToInt DoubleExpr
 and DoubleExpr = DoubleLVal LVal
              | DoubleLit real
              | DoubleFunCall Identifier ExprList
              | DoubleUMinus DoubleExpr
              | DoubleArith ArithOp DoubleExpr DoubleExpr
              | DoubleCond Expr DoubleExpr DoubleExpr
              | IntToDouble IntExpr
  and RefExpr = RefNull
              | RefLVal LVal
              | RefAddr LVal
              | RefCond Expr RefExpr RefExpr
              | RefFunCall Identifier ExprList
              | RefFromVoid RefExpr Type
              | RefToVoid RefExpr
  and Expr = IntExpr IntExpr
              | DoubleExpr DoubleExpr
              | RefExpr RefExpr
  and ExprList = ExprList_Nil
              | ExprList_Cons Expr ExprList
```

Figure 5.1: Abstract syntax of C expressions in Isabelle/HOL

isomorphic, but additionally includes the constructor *IVoid* for type **void**.[3] Expressions are built over the simplified lvalues already described in Sec. 3.3.1 (datatype *LVal*). All lvalues are labelled with their respective types, from which the types of expressions can in principle be inferred. In practice, however, the type information is only used for pointer dereferencing, array access, field selection, pointer difference, and casts to and from **void** *, where all but the last two work on the level of lvalues. In contrast to lvalues, which can be of any *Type*, we only distinguish between three types of expressions, namely integer, double and pointer (or reference) expressions. They correspond to the value domains *DomInt*, *DomDouble*, and *Loc* — since no distinction is made between different sizes of integers or floating-point numbers in the semantics, there is no use in distinguishing them on the level of abstract syntax. Integer expressions include function calls with an integer result (*IntFunCall*) and the arithmetic operations over integers (*IntUMinus*, *IntArith*) that are not related to bit-level operations. Further constructors represent conditional expressions (*IntCond*) of the form (e) ? ie1 : ie2, general comparisons of integer or double values (*IntIntComp*, *IntDoubleComp*) as well as equality comparisons between pointer expressions (*IntRefEq*), and the logical operators negation ! (*IntLNot*), conjunction && (*IntLAnd*), and disjunction || (*IntLOr*). Pointer subtraction (*IntRefDiff*) depends on type information, because pointer difference is expressed as a factor of the size of the pointed-to type, which is why the corresponding constructor expects the type of the objects that the two pointers point to. All conversions from floating-point expressions to integer expressions (and vice versa) are made explicit in the abstract syntax (via constructors *DoubleToInt* and *IntToDouble*).

Double expressions are defined analogously to integer expressions, with the difference being that they do not include the logical (effectively boolean) expressions nor pointer arithmetic. There is only a very limited number of pointer expressions: the constant NULL expression (*RefNull*); the use of an lvalue of pointer type (*RefLVal*); the address-of operator applied to an lvalue (*RefAddr*); conditional expressions and function calls as for integer and double expressions; and finally the conversion of an arbitrary pointer to and from type **void** * (*RefToVoid* and *RefFromVoid*). We did not include an explicit addition between a pointer and an integer, because this can be encoded via the equality between p + n and &p[n], i. e., we can add to a pointer by taking the address of an appropriate array access operation. This encoding favours iteration via array indices over iteration via pointer traversal: the following snippet can be verified,

```
char *s = "some␣string", *p;
for (p = s; *p; ++p) { /* use of p here */ }
```

but the encoding of ++p as p = &p[1] will make it more laborious than the equivalent

```
char *s = "some␣string"; int i;
for (i = 0; s[i]; ++i) { /* use of s[i] here */ }
```

Finally, datatypes *Expr* and *ExprList* allow for a uniform treatment of expressions of different types. This way we can subsume function calls with different

[3] In the memory model, there is no use for this type, as no object of type void exists. We will denote the natural translation (ignoring void) from *Type* to *RTT* with a subscript ϑ, as in $(\mathit{BasicType\ IInt})_\vartheta = \mathit{RTT\text{-}bas\ BR\text{-}Int}$.

concrete argument types under a single constructor determined by the return type, e. g. *IntFunCall*.

5.2.2 Statements and Declarations

Fig. 5.2 presents statements and declarations. The former (datatype *Stmt'*) are parameterised by two type variables *'a* and *'b* that serve as placeholders for specific annotations. An *EmptyStmt* represents the empty statement (;). The binary *ConditionalStmt* represents **if** statements and hence expects an expression (which is given the 'not-null-or-zero' boolean interpretation as in C) and two statements, representing the *then* and *else* branches. Since all loops are normalised, we only require a single constructor for **while** loops (*WhileStmt*), expecting an expression, a statement for the body and a value of type *'b* interpreting the @invariant, @modifies and @variant clauses of loop annotations. This parameterisation allows us to substitute the concrete type for loop annotations without having to modify definitions or theorems that do not relate to these. Concretely, we experimented with both deeply and shallowly embedded loop annotations, where we finally opted for the latter (cf. datatype *LoopAnno* below). Constructor *SpecStmt* is used for statement specifications. The type parameter *'a* must thus be instantiated with a datatype representing the @join and @modifies annotations, e. g. *StmtAnno* below. Unlike in C, assignments are statements (constructor *AssignStmt*), which is in accordance with our requirement that they only occur at the statement level. The left-hand side of an assignment has to be an lvalue, while the right-hand side can be an arbitrary expression. If the value of a function call is discarded (i. e., if it appears at the statement level), we christen it a procedure call (*ProcCallStmt*). Other expressions may also be evaluated only for the sake of their side-effects by converting them into statements via *ExprStmt*. *SeqStmt* concatenates two statements, thus achieving a uniform treatment of compound and atomic statements. C labels are modelled by a *LabeledStmt* that simply expects an identifier and a statement, which may be compound. All labels within a function must have distinct names. The verification environment uses labels only within specifications to refer to the values of expressions at particular program points, and not for unstructured jumps. Labels are therefore interpreted hierarchically: given *LabeledStmt* l s, label l is only visible in the (compound) statement s. We thus ensure that all labels occurring in a specification expression attached to a statement s refer to program points that have been passed on the way to s.

As discussed above, five statements are missing compared to the C grammar: the unstructured **switch** statement and the four jump statements **break**, **continue**, **return**, and **goto**. The single exit requirement is enforced by introducing an independent *ReturnStmt* datatype, which gets attached to a *Stmt'* in the datatype *Block'*, which is used as the datatype for function bodies. It additionally contains a list of variable declarations (*Vardecl*), which neatly enforces the declaration of all function-local variables at the beginning of the function.

At the level of global declarations and definitions we must distinguish between variables and functions. Both variable declarations and definitions are represented as a *Vardecl* which contains an identifier, a type, and an optional *Initialiser*. The latter is simply a rose tree of expressions, built in concrete syntax through the use of braced initialisers. Function declarations are represented by *Funheader'*, parameterised once again over a type variable *'a* used for func-

```
datatype (’a, ’b) Stmt’ = EmptyStmt
          | ConditionalStmt Expr "(’a, ’b) Stmt’" "(’a, ’b) Stmt’"
          | WhileStmt Expr "(’a, ’b) Stmt’" "’b"
          | SpecStmt "(’a, ’b) Stmt’" "’a"
          | AssignStmt LVal Expr
          | ProcCallStmt Identifier ExprList
          | ExprStmt Expr
          | SeqStmt "(’a, ’b) Stmt’" "(’a, ’b) Stmt’"
          | LabeledStmt Identifier "(’a, ’b) Stmt’"

datatype ReturnStmt = ReturnStmt "Expr option"

datatype Paramdecl = Paramdecl Identifier Type

datatype Initialiser = InitialVal Expr
                     | Initialisers "Initialiser list"

datatype Vardecl   = Vardecl Identifier Type "Initialiser option"

datatype (’a, ’b) Block’ =
               Block "Vardecl list" "(’a, ’b) Stmt’" ReturnStmt

datatype ’a Funheader’ = Funheader Identifier "Paramdecl list" "’a"

datatype (’a, ’b, ’c) Fundef’ =
               Fundef "’a Funheader’" "(’b, ’c) Block’"

datatype (’a, ’b, ’c) Decl’ = FunDecl "’a Funheader’"
                            | FunDef  "(’a, ’b, ’c) Fundef’"
                            | GVardecl Vardecl

datatype LoopAnno = LoopAnno "(Env ⇒ unit SP)"
                             "(MVal list) option"
                             "(Env ⇒ State ⇒ nat) option"

datatype StmtAnno = StmtAnno "Env ⇒ unit SR" "MVal list"

datatype FunSpec = FunSpec "(Env ⇒ Val list ⇒ unit SP)"
                           "(Env ⇒ Val list ⇒ (Val option) SR)"
                           "(MVal list) option"


types Stmt = "(StmtAnno, LoopAnno option) Stmt’"
      Block = "(StmtAnno, LoopAnno option) Block’"
      Funheader = "(FunSpec option) Funheader’"
      Fundef = "(FunSpec option, StmtAnno, LoopAnno option) Fundef’"
      Decl = "(FunSpec option, StmtAnno, LoopAnno option) Decl’"
      TranslationUnit = "Decl list"
```

Figure 5.2: Abstract syntax of C statements and declarations in Isabelle/HOL

tion specifications. The definition of a function (*Fundef'*) additionally contains a body in terms of a *Block'*; hence, it is triply parameterised over its function specification and the loop and statement annotations in the body. Datatype *Decl'* summarises all three external declarations that can occur at file scope.

The concluding type definitions in Fig. 5.2 instantiate the type parameters of all primed datatypes with concrete specifications and annotations, yielding the types *Stmt*, *Block*, etc., that are actually used in the verification environment. In particular, a *TranslationUnit* is just a list of external declarations. The meaning of the respective datatypes *LoopAnno*, *StmtAnno*, and *FunSpec* will only be explained in Sec. 5.5 as they involve as yet unknown types for evaluation environments and state predicates. Intuitively, a loop annotation contains the semantics of the loop invariant (since we use a shallow embedding of specifications, the invariant is not represented as an explicit datatype), an optional list of mvalues representing the modifies clause, and the optional semantics of the variant. Likewise, a *FunSpec* consists of the semantics of the precondition and postcondition, and an optional list of mvalues.

5.2.3 Translation Units and Linked Programs

C implements a rather simple concept of modularity in terms of *translation units* [82, § 5.1.1.1]. A translation unit is the result of applying the C preprocessor to a single source file, i. e., it is the source file with all **#include**d files spliced in and all **#define**d symbols substituted. Translation units can be compiled separately, and later linked together to form an executable program. The provided external interface of a translation unit, i. e. the set of all identifiers that are visible to other translation units and for which storage is reserved, is given by all identifiers that are *defined* with *external linkage*. Whether a file scope identifier has external linkage depends on its declaration: Ignoring multiple declarations, variables declared with no storage class specifier have external linkage, just as those declared with the **extern** specifier. Variables declared as **static** have internal linkage and are hence invisible to other units. Functions have external linkage, unless they are specified with **static**. To complicate matters, the concept of linkage does not yet mandate which translation unit actually reserves storage for the object denoted by an external identifier. There must be exactly one translation unit in which the identifier is actually defined. The *definition* of a variable identifier is caused either by a declaration that has an initialiser or by one that does not include the **extern** specifier. Matters are further complicated by the fact that multiple declarations of identifiers with the same name can hardly be avoided in C, due to the lack of a proper namespace concept: including a header file of some library might introduce external declarations of identifiers whose names are also used for different purposes in the translation unit at hand. To avoid having to know about all names introduced by such header files, a peculiar special case has been introduced: if an identifier gets declared with external linkage, but has previously been declared with internal linkage, then the linkage is not altered, and the second declaration is effectively ignored. This makes the following code snipped be legal C, in which print_int will output 0:

```
static int c = 0;
extern int c;
```

```
int main(void) { print_int(c); return 0;}
```

This suggests a model of a *normalised* translation unit, consisting of a sequence of external identifiers whose storage is defined outside the unit (but whose names and types are known), followed by a sequence of external identifiers whose storage is defined inside the unit and which are visible to other units, followed by a sequence of identifiers whose storage and visibility are restricted to the unit itself. Such a normalisation is also performed by the CIL C front-end [113].

Translation units do not constitute parts of a program that can be executed in isolation, due to the lack of storage for some external identifiers. The same holds for the verification of a program function, which also requires that all variables occurring in it have memory (in our case a base location) assigned to them. Therefore, a translation unit is not an appropriate concept for modular verification. Instead, we assume that all identifiers used by a function will eventually be assigned storage by *some* translation unit when the whole program is linked together. We therefore verify each function in a context in which all variables used by the function exist, and have their corresponding base location assigned to them.

In Sec. 4.2.4 we assigned global identifiers a base location identified purely by name. Since each identifier with external linkage may only be defined exactly once, we can assume a memory location *Global* "x" for each identifier x declared with external linkage. For identifiers with internal linkage, we need to distinguish between an x declared in translation unit T_1, e. g. as **static int** x, and another x from T_2 declared as **static double** x. The solution is to introduce unique identifiers for translation units (denoted by $id(T_i)$ below), and assign each x of T_i the base location

$$\mathit{Global}\ \text{"}\langle id(T_i)\rangle \dagger x\text{"}$$

where † is some symbol not allowed inside regular C identifiers. We thereby lift all external identifiers to the same global visibility level. It is only a theoretical problem that this allows to syntactically refer to variables with internal linkage from functions in other translation units: programs are always translated from C source code, and never written explicitly as Isabelle/HOL datatypes. In summary, this approach allows for modularity in the sense that each function can be verified in isolation, under the assumption that it will eventually be embedded in a program that assigns memory to all occurring identifiers. Functions appearing in different translation units will correctly refer to the same memory location when using equal identifiers with external linkage, while referring to different memory locations when using equal identifiers with internal linkage.

5.3 Semantics

We defined a denotational semantics for C programs, against which the program logic of the next chapter is proven correct. The semantics is denotational instead of operational for two reasons:

- It is symbolically executable by Isabelle's simplifier, because it is defined by a set of unconditional equations. We can make good use of this fact during verification condition generation to ameliorate the situation of not being able to directly use program terms within specifications. Instead, we use the *semantics* of these expressions, and let the simplifier reduce them to terms that are nearly equal to what other approaches achieve by a direct use of expressions. An interesting way of *validating* the semantics by comparing its evaluation results on concrete inputs with test data obtained from feeding the same input to compiled programs has been described in [24]. Such a validation can only be done with an executable semantics.

- We are at any rate interested in a deterministic semantics; operational semantics usually require determinism proofs, due to the non-deterministic nature of the rule-based formulation. In contrast, a denotational semantics can be easily formulated so as to make determinism inherent.

We introduce type definitions for state predicates, state relations, and for state transformers which are used in the subsequent definitions.

```
types
  'a SP  = "'a ⇒ State ⇒ bool"
  'a SR = "State ⇒ ('a × State) ⇒ bool"
  'a ST = "State ⇀ ('a × State)"
```

A *state predicate* (*SP*) is a predicate over a *State* and an additional parameter *'a*; a *state relation* (*SR*) is similar, but expects two states. *State transformers* (*ST*) form the basis of the denotational semantics: they are partial functions from a *State* to a *result* of type *'a* and a successor *State*. Note that this is exactly the type of a *state monad with exceptions* [110], allowing us to use the well-known monadic operations *bind* (written ≫=), *eta*, and *failure*:

```
definition
  bind  :: "'a ST ⇒ ('a ⇒ 'b ST) ⇒ 'b ST"
where
  "bind f g = (λS. case f S of
                    None ⇒ None
                  | Some (a, S') ⇒ g a S')"
definition
  "eta x = (λs. Some (x, s))"
definition
  "failure = (λs. None)"
```

Operator ≫= sequences two monadic operations, where the right-hand side operand g is parameterised over the result of the left-hand side f. It yields a monadic operation that first evaluates f and then, in the case of success, applies g to the result a and successor state S'. Failure is propagated, i. e., if f fails (by yielding *None*), g is not evaluated at all. *eta* simply injects a value into the monad, leaving the state untouched, while *failure* is the monadic operation that always fails. All failures are treated identically as *total* failures, and there is no *catch* operation to recover from a failure.

5.3.1 Evaluation Context

Programs, statements, expressions, and lvalues are evaluated w. r. t. an environment Γ :: *Env*, defined as an Isabelle/HOL record which contains three sub-environments in terms of a mapping from local and global variables to their locations, one from labels to program states and one from function identifiers to the function's signature (including its specification) as well as its semantics.

```
types
   FunSem = "(Val list ⇒ (Val option) ST)"
record FunData =
   FDSpec :: "(Val list ⇒ unitSP) × (Val list ⇒ (Val option) SR)"
   FDMod  :: "Val list ⇒ State ⇒ Loc set"
   FDSig  :: "Paramdecl list × Type"
record Env =
   funEnv  :: "Identifier ⇀ (FunData × FunSem option)"
   varEnv  :: "Identifier ⇀  Loc"
   labEnv  :: "Identifier ⇀ State"
```

The denotational semantics of C functions have type *FunSem*; they are state transformers whose result is the function's return value (which is *None* precisely if the C function has return type **void**) that are parameterised over the actual function arguments (of type *Val list*). Record *FunData* contains the semantics of the precondition and postcondition of a C function in its *FDSpec* field (*Val list* ⇒ *unitSP* and *Val list* ⇒ (*Val option*) *SR*, respectively). Their signatures convey their assigned semantics. Both are parameterised over the actual function arguments. The precondition is a state predicate ignoring the additional parameter (by instantiating it with type *unit*), while the postcondition is a state relation which may refer to values of expressions in the post-state as well as the pre-state. Its *Val option* argument stands for the optional function result value. The semantics of a @modifies clause (*FDMod*) is a function yielding the set of locations that may be modified by the function, given the function's arguments (*Val list*) and the pre-state. It cannot simply be a *Loc set* since a @modifies clause is state dependent, as it may contain dereference operations, e. g., *p for a global variable p of pointer type. Finally, field *FDSig* contains the types and names of the function parameters as well as the return type.

5.3.2 Denotational Semantics

Lvalues and Expressions

Since expressions may have side-effects, we assign to each lvalue and expression datatype a semantic function that yields a state transformer with a result type denoting the value of the respective term. Lvalues evaluate to the location they describe, integer expressions evaluate to *DomInt*, and so forth:

```
sem_lv       :: "Env ⇒ LVal ⇒ Loc ST"
sem_int      :: "Env ⇒ IntExpr ⇒ DomInt ST"
sem_double   :: "Env ⇒ DoubleExpr ⇒ DomDouble ST"
sem_ref      :: "Env ⇒ RefExpr ⇒ Loc ST"
sem_expr     :: "Env ⇒ Expr ⇒ Val ST"
sem_exprlist :: "Env ⇒ ExprList ⇒ (Val list) ST"
```

The semantics of general expressions simply delegate the evaluation to the corresponding type of expression and wrap the result by the corresponding constructor of datatype *Val*:

```
"sem_expr Γ (IntExpr e)    = (sem_int Γ e >>= eta ∘ IntVal)"
"sem_expr Γ (DoubleExpr e) = (sem_double Γ e >>= eta ∘ DoubleVal)"
"sem_expr Γ (RefExpr e)    = (sem_ref Γ e >>= eta ∘ PtrVal)"
```

The semantic function for lvalues must take care that no invalid memory accesses occur. Such accesses are possible both for array accesses and dereferencing, in which case the denotation must be a failure. Plain identifiers and structure field selection are always safe; the former are simply looked up in the environment Γ, while the latter first evaluate their left-hand side lvalue, bind the result and apply the field selection function of the memory model to it. An array access first evaluates the reference expression r denoting the array (binding it to l), then evaluates the index expression to i, and then performs two checks: *nullchk* ensures that l is not null (failing otherwise), while *arraychk* fails if accessing l at index i will exceed the array bounds or if l does not denote an array element itself at all. Note that no distinction is made whether r is of array or pointer type: C's pointer decay is elegantly avoided through the fact that all array variables point to their initial element in our memory model (cf. array head pointers in Sec. 4.2.3). Dereferencing works similar to array access, by first evaluating the reference expression and then checking its non-nullity and its validity (*defchk*).

```
"sem_lv Γ (LId i t)  = eta (lookup_var Γ i)"
"sem_lv Γ (LFieldSel lv i t) = (sem_lv Γ lv >>= λl.
                                 (eta (l(type_lv lv)ϑ→i)))"
"sem_lv Γ (LArrayAcc r e t) = (sem_ref Γ r >>= λl.
                               (sem_int Γ e >>= λi.
                               (nullchk l >>= λl.
                               (arraychk l (tϑ) i))))"
"sem_lv Γ (LDeref r t) = (sem_ref Γ r >>= nullchk >>= (λl.
                                 defchk l (tϑ)))"
```

The semantics of integer, double and pointer expressions are rather straightforward. Due to the restriction that all expressions must yield a deterministic value under all evaluation orders or fail otherwise, we can fix a left-to-right evaluation order. We briefly describe the equations for reading an lvalue, for function calls and for binary arithmetic operations.

```
"sem_int Γ (IntLVal lv) = (sem_lv Γ lv >>= read_int_st)"
"sem_int Γ (IntFunCall f es) = (sem_exprlist Γ es >>=
                                ((lookup_funsem Γ f)) >>=
                                eta ∘ valToInt ∘ the)"
"sem_int Γ (IntArith arop e1 e2) =
        (sem_int Γ e1 >>= λv1.
        (sem_int Γ e2 >>= λv2.
          eta (arith_op arop v1 v2)))"
```

An lvalue interpreted as an integer expression is evaluated by first obtaining the location it denotes through *sem-lv* and then applying *read-int* to it.[4] Function calls are evaluated by first evaluating all function arguments, then looking up the semantics of the function by its name in the environment Γ, and finally aggressively extracting the integer return value from the function result of type *Val option.* A binary arithmetic expression first evaluates its operands in sequence and then applies the appropriate operator obtained via *arith-op* to the resulting values.

Double expressions and pointer expressions are evaluated in a similar fash-

[4] The monadic notation forces us to introduce a side-effect free operation *read-int-st* which lifts *read-int* to the monadic level.

```
fun
 iter :: "bool ST ⇒ unit ST ⇒ nat ⇒ bool ST"
where
  "iter b p 0 = eta True"
| "iter b p (Suc n) = (b >>= (cond (p >> iter b p n) (eta False)))"
definition
  loop_cond :: "bool ST ⇒ unit ST ⇒ nat ⇒ State ⇒ bool"
where
  "loop_cond b p n x = (∃z. iter b p n x = Some (False, z))"
definition
  iter_sem :: "bool ST ⇒ unit ST ⇒ bool ST"
where
  "iter_sem b p x = (if (∃ n. loop_cond b p n x)
                     then iter b p (LEAST n. loop_cond b p n x) x
                     else None)"

fun
  sem_s :: "Env ⇒ Stmt ⇒ unit ST"
where
  "sem_s Γ (EmptyStmt) = skip"
| "sem_s Γ (ConditionalStmt c p q) =
      (sem_bool Γ c >>= cond (sem_s Γ p) (sem_s Γ q))"
| "sem_s Γ (WhileStmt b p _) =
      (iter_sem (sem_bool Γ b) (sem_s Γ p) >> skip)"
| "sem_s Γ (AssignStmt v e) = (sem_lv Γ v >>= λlv.
                              (sem_expr Γ e >>= λev.
                               upd lv ev))"
| "sem_s Γ (ProcCallStmt idt args) =
      (sem_exprlist Γ args >>= ((lookup_funsem Γ idt)) >>= skip'))"
| "sem_s Γ (ExprStmt e) = (sem_expr Γ e >>= skip')"
| "sem_s Γ (LabeledStmt l stmt) = (λS. sem_s (add_label Γ l S) stmt
S) "
| "sem_s Γ (SeqStmt s1 s2) = (sem_s Γ s1 >>= λ_. sem_s Γ s2)"
| "sem_s Γ (SpecStmt s spec) = (sem_s Γ s)"
```

Figure 5.3: Semantics of statements

ion. For example, the semantics of the address-of operator evaluates the lvalue operand and simply yields the obtained location as the result. Conversions from a pointer type to type **void** ∗ and vice versa are semantically transparent: the corresponding equations simply delegate the evaluation to the operand expression. This is possible because all locations have the same type in the memory model. The ‘dynamic’ type check occurs whenever a pointer is dereferenced, as explained above.

```
"sem_ref Γ (RefAddr lv) = (sem_lv Γ lv >>= eta)"
"sem_ref Γ (RefFromVoid re t) = (sem_ref Γ re)"
```

Statements

The semantics of statements are presented in Fig. 5.3. Most equations are standard. In a conditional statement the condition c is evaluated as a boolean value (non-null pointers and non-zero integers and doubles yield *True*) and depending on the outcome, the then- or else-branch is evaluated.[5] The denotation of loops is usually expressed in terms of a least-fixed point operator over the domain of state transformers; this is however not necessary and an equivalent, more intuitive and operational definition can be given in terms of the bounded iteration operator *iter* [84]. The definition is complicated by the fact that loop conditions may have side-effects. A while loop evaluates its loop condition (*sem-bool* Γ b) and its body (*sem-s* Γ p) in turn until the condition becomes *False*. If this never happens, the loop diverges, which we identify with failure because we are only interested in terminating programs. *iter* $b\ p\ n$ (with b and p state transformers) gives the semantics of a while loop bounded to n iterations. Its boolean result indicates the final outcome of the loop condition. Predicate *loop-cond* $b\ p\ n\ \sigma$ tells whether the loop terminates after at most n iterations starting in σ, which is the case if the result of *iter* $b\ p\ n\ \sigma$ is *False*. The overall semantics of a while loop are now given by *iter-sem*: it is the n-fold iteration via *iter* such that n is the least number of iterations after which the loop condition becomes *False*, if such a number exists. Otherwise, the semantics is identified with failure.

The litmus test for a correct definition of the semantics of loops is the *unfolding lemma*:

```
lemma while_unfold:
  "sem_s Γ (WhileStmt c s a) =
  sem_s Γ (ConditionalStmt c (SeqStmt s (WhileStmt c s a)) EmptyStmt)"
```

Evaluating a loop is the same as testing the loop condition, and executing the loop body once followed by the whole loop in case it evaluates to *True*, and doing nothing otherwise.

The other noteworthy equation is that for a labelled statement, which has an effect on the environment that its statement operand is evaluated in. It simply adds a mapping from the given label to the current state into the environment.

Declarations

A complete formalisation requires further definitions for the return statement, for blocks, local variable declarations, and variable initialisation. We elide their definitions, as they are merely of an infrastructural character. The semantics for function definitions is interesting, as it differs from similar formalisations such as [136]. Recall that functions are state transformers parameterised over a list of actual argument values. The semantics of function calls therefore becomes trivial, since the function semantics simply has to be looked up in the environment. No action has to be taken that prepares a function call, such as writing the argument values to special ‘parameter passing’ locations in memory. Likewise, the semantics of functions does not involve fetching the arguments. However, since function parameters are ordinary variables that may be modified within the body of the function, they must be available as memory locations. This is achieved by using a higher-order operator that realises *local scoping* of

[5] *cond* $a\ b\ c$ is equal to *if* c *then* a *else* b and is only used for notational convenience.

a location: *local-loc* $p\ x\ t$ allocates an arbitrary fresh memory location l with name x and type t, calls $p\ l$, and afterwards deallocates the location. The result of this operation is that of $p\ l$.

```
definition
  local_loc :: "(Loc ⇒ 'a ST) ⇒ string ⇒ RTT ⇒ 'a ST"
where
  "local_loc p x t = (fresh_loc_st x t >>= λl.
                      (p l        >>= λr.
                      (dealloc_st l >>
                       eta r)))"
```

This behaviour is exactly what is required for the semantics of functions: Given the function body and non-empty lists of argument values and parameters, in *sem-paramdecls* we introduce a local scope for the first parameter with *local-loc*, passing it a state transformer that immediately updates the state at the freshly generated location *loc* with the argument value and continues with the remaining parameters in an updated environment that maps the parameter name to *loc*. Finally, when all parameters are processed, the so extended state and environment are used to evaluate the function body via *sem-block*.

```
fun
  sem_paramdecls ::
  "Env ⇒ Block ⇒ Paramdecl list ⇒ Val list ⇒ (Val option) ST"
where
  "sem_paramdecls Γ blk [] [] = (sem_block Γ blk)"
| "sem_paramdecls Γ blk (p#ps) (v1#vs) =
  (case p of
    Paramdecl name typ ⇒
      local_loc (λloc. (update_st loc v1 >>= λ_.
        sem_paramdecls (add_var Γ name loc) blk ps vs)) name (typ_ϑ))"
fun
  sem_fundef    :: "Env ⇒ Fundef ⇒ Val list ⇒ (Val option) ST"
where
 "sem_fundef Γ (Fundef hdr blk) args =
    sem_paramdecls Γ blk (funParams hdr) args"
```

The semantics of a function is then given by *sem-fundef* which merely delegates to *sem-paramdecls* after extracting the parameters from the function definition.

Since we disallow recursion, the call graph of any admissible program is a tree. Therefore, the set of all its functions can be linearised in such a way that that no function at index i will call, directly or through other functions, any function with an index $\geq i$. The linearisation allows us to build an explicit environment for every function in which its semantics is evaluated. Such an environment contains the semantics of all functions appearing earlier in the linearisation (cf. Sec. 6.2).

5.4 Side-effect Free Expression Evaluation

By using the standard monad laws and by unfolding the definition of the monadic read operations:

$$(\mathit{eta}\ x \gg= g) = g\ x \tag{eta-lunit}$$
$$(f \gg= \mathit{eta}) = f \tag{eta-runit}$$
$$f \gg= (\lambda x.\ g\ x \gg= h) = ((f \gg= g) \gg= h) \tag{bind-assoc}$$
$$\mathit{read\text{-}int\text{-}st}\ l\ \sigma = \mathit{Some}\ (l\ @_i\ \sigma,\ \sigma) \tag{read-int-st}$$

the semantics of simple expressions can be evaluated by Isabelle's simplifier; for example, in an environment Γ mapping x and y to local variables $\nu{\cdot}x_t$ and $\nu{\cdot}y_t$ we obtain the following for the expression x + y:

$$\begin{array}{ll}
\mathit{sem\text{-}int}\ \Gamma\ (\mathit{IntArith\ OpPlus}\ (\mathit{IntLVal}\ (\mathit{LId}\ x\ t)) & \\
\qquad\qquad\qquad\qquad\qquad (\mathit{IntLVal}\ (\mathit{LId}\ y\ t)))\ \sigma & = \\
\mathit{read\text{-}int\text{-}st}\ \nu{\cdot}x_t \gg= \lambda v_1. & \\
\quad \mathit{read\text{-}int\text{-}st}\ \nu{\cdot}y_t \gg= \lambda v_2. & \\
\quad\quad \mathit{eta}\ (v_1 + v_2) & = \\
\mathit{Some}\ (\nu{\cdot}x_t\ @_i\ \sigma + \nu{\cdot}y_t\ @_i\ \sigma,\ \sigma) &
\end{array}$$

The initial and final state coincide and the syntactic addition has been rewritten to a semantic addition. The latter is a desirable property for program verification, since we obtain a direct correspondence between program expressions and semantic terms. Expressions in which the program state is merely read and never written to and which do not result in failure occur frequently, particularly as conditional expressions of **if** and **while** statements. Unfortunately, allegedly simple expressions that dereference a pointer or access an array may result in failure, so that these expressions cannot be simplified in a useful way. Consider the evaluation of *p for a local variable p:

$$\begin{array}{ll}
\mathit{sem\text{-}int}\ \Gamma\ (\mathit{IntLVal}\ (\mathit{LDeref}\ (\mathit{RefLVal}\ (\mathit{LId}\ p\ t))\ t'))\ \sigma & = \\
(\mathit{read\text{-}loc\text{-}st}\ (\nu{\cdot}p_t) \gg= \lambda x. & \\
\quad (\mathit{nullchk}\ x \gg= \lambda x'.\ (\mathit{defchk}\ x'\ t_\vartheta \gg= \mathit{read\text{-}int\text{-}st})))\ \sigma &
\end{array}$$

To simplify the right-hand side further, rewrite rules related to *nullchk* and *defchk* are required. However, such rules can be formulated much more easily in an extensional fashion and not inside the monad (i. e., we need to make the state explicit):

$$\mathit{valid\text{-}loc}\ l\ t\ \sigma \Longrightarrow \mathit{defchk}\ l\ t\ \sigma = \mathit{eta}\ l\ \sigma \wedge \mathit{nullchk}\ l\ \sigma = \mathit{eta}\ l\ \sigma$$

We therefore do not simplify expressions by using the monadic semantics, but follow a different strategy in which we define a simpler semantics for the evaluation of side-effect free expressions, whose equivalence w. r. t. the full monadic semantics we prove under the assumption that the evaluated expression is in fact side-effect free.

First of all, we require a formal notion of side-effect freeness for all types of expressions. An expression is *side-effect free* for an environment Γ and a state σ if its evaluation under Γ and σ does not alter σ:

definition

```
sef_expr :: "Env ⇒ Expr ⇒ State ⇒ bool"
where
"sef_expr Γ e σ = (∃v. sem_expr Γ e σ = Some (v, σ))"
```

Note that this a semantic criterion that allows an expression to alter the state temporarily, as long as the initial state is finally restored.

We then partially define a family of semantic functions that merely read the state to yield a value, ignoring all possible failures that might occur during evaluation:

```
sef_sem_lv       :: "Env ⇒ LVal ⇒ State ⇒ Loc"
sef_sem_int      :: "Env ⇒ IntExpr ⇒ State ⇒ DomInt"
sef_sem_double   :: "Env ⇒ DoubleExpr ⇒ State ⇒ DomDouble"
sef_sem_ref      :: "Env ⇒ RefExpr ⇒ State ⇒ Loc"
sef_sem_expr     :: "Env ⇒ Expr ⇒ State ⇒ Val"
```

These functions are defined via primitive recursion over the expression datatypes. They are under-specified because we do not provide equations for function calls, since these are generally not side-effect free. The equations for constructors whose semantics is naturally side-effect free are rather obvious. An lvalue is read as an integer by evaluating the lvalue as a location and reading at that location; pointer equality can be translated to a direct comparison between two recursively evaluated pointer values, which is then converted to *DomInt*:

```
"sef_sem_int Γ (IntLVal lv) =
  (λσ. (sef_sem_lv Γ lv σ) @_i σ)"
"sef_sem_int Γ (IntRefEq r1 r2) =
  (λσ. bool_to_int (sef_sem_ref Γ r1 σ = sef_sem_ref Γ r2 σ))"
```

The interesting cases are those for array access and pointer dereferencing, where the sanity checks of the full monadic semantics are simply omitted:

```
"sef_sem_lv Γ (LArrayAcc r e t) =
    (λσ. let l = (sef_sem_ref Γ r σ);
             i = sef_sem_int Γ e σ
         in l_{t_ϑ}.[nat i])"
"sef_sem_lv Γ (LDeref r t) = (λσ. sef_sem_ref Γ r σ)"
```

Under these simple semantics, the expression `*p` from above gets fully simplified:

$$\textit{sef-sem-int}\,\Gamma\;(\textit{IntLVal}\;(\textit{LDeref}\;(\textit{RefLVal}\;(\textit{LId}\;p\;t))\;t'))\;\sigma \\ = (\nu{\cdot}p_t\,@_l\,\sigma)\,@_i\,\sigma$$

5.4.1 A Syntactic Condition for Side-Effect Freeness

We would expect that the side-effect free semantics is 'equivalent' to the full monadic semantics for side-effect free expressions, in the following way:

$$\textit{sef-expr}\,\Gamma\;e\;\sigma \Longrightarrow \textit{sem-expr}\,\Gamma\;e\;\sigma = \textit{Some}\;(\textit{sef-sem-expr}\,\Gamma\;e\;\sigma,\;\sigma) \tag{5.1}$$

This necessitates a proof of semantic side-effect freeness whenever a term matching the left-hand side is supposed to be replaced by the right-hand side. However, the definition of *sef-expr* does not provide any hints about its proof. Ideally, a criterion for side-effect freeness would be defined along the structure of the expression. It should be reducible to *True* or *False* in all simple cases; e. g., expressions containing function calls should reduce to *False*, while expressions without pointers, arrays and functions (like x + 9 − 3) should yield *True*. For array accesses and pointers we would like the criterion to reduce to validity constraints about the respective pointer and array access in terms of *valid-loc* and *valid-array-acc*. We can define this criterion as follows:

```
syn_sef_lv      :: "Env ⇒ LVal ⇒ State ⇒ bool"
syn_sef_int     :: "Env ⇒ IntExpr ⇒ State ⇒ bool"
syn_sef_double  :: "Env ⇒ DoubleExpr ⇒ State ⇒ bool"
syn_sef_ref     :: "Env ⇒ RefExpr ⇒ State ⇒ bool"
syn_sef_expr    :: "Env ⇒ Expr ⇒ State ⇒ bool"
```

Some representative equations show how the condition for side-effect freeness in an environment Γ and a state σ is built up; literals are always side-effect free, the condition for lvalues and binary arithmetic operators is built by delegation, and function calls are (conservatively) regarded as not side-effect free.

```
"syn_sef_int Γ (IntLit i)  = (λσ. True)"
"syn_sef_int Γ (IntLVal lv) = (λσ. syn_sef_lv Γ lv σ)"
"syn_sef_int Γ (IntArith arop e1 e2) =
        (λσ. syn_sef_int Γ e1 σ ∧ syn_sef_int Γ e2 σ)"
"syn_sef_int Γ (IntFunCall i e) = (λσ. False)"
```

The interesting equations are those for array and pointer access. An array access is side-effect free if both the pointer expression denoting the array and the index expression are, and if the array access itself is valid according to *valid-array-acc*. Likewise, a pointer access is side-effect free if the expression being dereferenced is and if the pointer is valid according to *valid-loc*. In both cases we use the side-effect free semantics to evaluate the expressions used in the validity constraints:

```
"syn_sef_lv Γ (LArrayAcc r e t) =
   (λσ. syn_sef_ref Γ r σ ∧
        syn_sef_int Γ e σ ∧
       (let l = sef_sem_ref Γ r σ;
            i = sef_sem_int Γ e σ
        in •[l t_ϑ| i| σ]))"
"syn_sef_lv Γ (LDeref r t) =
   (λσ. syn_sef_ref Γ r σ ∧ •((sef_sem_ref Γ r σ) t_ϑ| σ))"
```

5.4.2 Equivalence of Semantics

Analogous to the proposition of Eq. (5.1) we can now establish the correctness of the side-effect free semantics:

theorem *sef_sem_correct:*

```
"syn_sef_expr Γ e σ ⟹
   sem_expr Γ e σ = Some (sef_sem_expr Γ e σ, σ)"
```

The proof is by induction over the structure of the expression datatypes and therefore requires the consideration of 37 cases. It is a good example of the usefulness of machine-assisted proofs, since the sheer number makes it easy to overlook cases or wrongly consider cases obvious. Actually, all but two out of 37 cases are proven automatically, where the two cases are the expected ones for array and pointer access. The proof goal for pointer access looks as follows after simplification:

$$\begin{array}{l}
[\![\ \mathit{syn\text{-}sef\text{-}ref}\,\Gamma\ r\ \sigma;\\
\ \ \bullet((\mathit{sef\text{-}sem\text{-}ref}\,\Gamma\ r\ \sigma)_{t_\vartheta}|\ \sigma);\\
\ \ \mathit{sem\text{-}ref}\,\Gamma\ r\ \sigma = \mathit{Some}\ (\mathit{sef\text{-}sem\text{-}ref}\,\Gamma\ r\ \sigma,\ \sigma)\ ]\!]\Longrightarrow\\
\ \ (\mathit{sem\text{-}ref}\,\Gamma\ r \ggg\!= \lambda x.\ (\mathit{nullchk}\ x \ggg\!= \lambda l.\ \mathit{defchk}\ l\ t_\vartheta))\ \sigma =\\
\ \ \mathit{Some}\ (\mathit{sef\text{-}sem\text{-}ref}\,\Gamma\ r\ \sigma,\ \sigma)
\end{array}$$

We may assume that the reference expression r is syntactically side-effect free and that it represents a valid location. Further, we know that both semantics coincide for r. We need to show that the additional checks *nullchk* and *defchk* both do not result in failure, do not modify the state, and simply pass the location computed by *sem-ref* Γ r through without altering it. This is ensured by the validity assumption for *sef-sem-ref* $\Gamma\ r\ \sigma$. The case for array access can be closed in a similar fashion.

Theorem *sef-sem-correct* trivially implies that the syntactic criterion for side-effect freeness implies the semantic one:

```
corollary syn_sef_expr_sound:
  "syn_sef_expr Γ e σ ⟹ sef_expr Γ e σ"
```

5.5 Specifications in Isabelle/HOL

In contrast to programs, specifications are embedded shallowly as Isabelle/HOL functions. We cannot define this translation within Isabelle/HOL over the abstract syntax of specification expressions (cf. Fig. 3.2) in terms of the semantic functions *sem-expr* etc., since arbitrary Isabelle/HOL code may appear in quotations, and anti-quotations may refer back to bound variables introduced in quotations. That is, while one would certainly expect the semantics of the specification expression

$$\texttt{\$\{}\ \ \exists y :: \mathit{DomInt}.\ y * y\ =\ \texttt{`\{x + 1\}}\ \ \texttt{\}}$$

to include $\exists y :: \mathit{DomInt}.\ y * y = l_x\ @_i\ \sigma + 1$ for appropriate σ and l_x, there is no way in Isabelle/HOL to reflect the fact that an identifier `$i` in abstract syntax refers to a quantified variable, as in

$$\texttt{\$\{}\ \ \exists i\ y :: \mathit{DomInt}.\ y * y\ =\ \texttt{`\{x[\$i]\}}\ \ \texttt{\}}$$

Therefore, the translation from the specification language to Isabelle/HOL expressions is performed by the front-end, which outputs the latter as plain strings to be parsed and analysed by Isabelle/HOL.

The types for preconditions, postconditions, invariants and statement annotations are specified by the constructors *FunSpec*, *LoopAnno*, and *StmtAnno* in Fig. 3.2. We focus on the translation of postconditions, as all other translations are similar and slightly simpler. A postcondition has type

$$\mathit{Env} \Rightarrow \mathit{Val\ list} \Rightarrow (\mathit{Val\ option})\ \mathit{SR}$$

The translation is performed by a collection of operations $\{\#^\sigma, \#^\sigma_{lv}, \#^\sigma_i, \#^\sigma_d, \#^\sigma_l\}$ on the abstract syntax of specification terms, for predicates, lvalues and the

Predicates:

$$
\begin{array}{rcl}
\#^{\sigma}(\mathsf{\backslash true}) & \overset{def}{=} & \mathit{True} \\
\#^{\sigma}(\mathsf{A\ \&\&\ B}) & \overset{def}{=} & (\#^{\sigma}(\mathsf{A}) \wedge \#^{\sigma}(\mathsf{B})) \\
\#^{\sigma}(\mathsf{\backslash result}) & \overset{def}{=} & f\,(\mathit{the}\ \Omega) \\
 & & \qquad (f \in \mathit{valTo}\{\mathit{Int}, \mathit{Double}, \mathit{Loc}\}) \\
\#^{\sigma}(\mathsf{\backslash forall\ \_Int\ x;\ e}) & \overset{def}{=} & \forall x :: \mathit{DomInt}.\ \#^{\sigma}(\mathsf{e}) \\
\#^{\sigma}(\mathsf{\backslash valid(p)}) & \overset{def}{=} & \bullet((\#^{\sigma}_{r}(\mathsf{p}))_{\mathit{type\text{-}of}(\mathsf{p})}|\ \sigma) \\
\#^{\sigma}(\mathsf{\backslash array(p,\ n)}) & \overset{def}{=} & \bullet[(\#^{\sigma}_{r}(\mathsf{p}))_{\mathit{type\text{-}of}(\mathsf{p})}|\ (\#^{\sigma}_{i}(\mathsf{n}))|\ \sigma] \\
\#^{\sigma}(\mathsf{E1} < \mathsf{E2}) & \overset{def}{=} & \#^{\sigma}_{x}(\mathsf{E1}) < \#^{\sigma}_{x}(\mathsf{E2}) \quad (x \in \{l, d\}) \\
\#^{\sigma}(\mathsf{\$f(e1,\ ...,\ eN)}) & \overset{def}{=} & f\ (\#^{\sigma}(\mathsf{e1}))\ \cdots\ (\#^{\sigma}(\mathsf{eN})) \\
\#^{\sigma}(\mathsf{\hat{}f(e1,\ ...,\ eN)}) & \overset{def}{=} & f\ \sigma\ (\#^{\sigma}(\mathsf{e1}))\ \cdots\ (\#^{\sigma}(\mathsf{eN})) \\
\#^{\sigma}(\mathsf{\$\{}s_1\ \mathsf{`\{a1\}}\ s_2\ \mathsf{`\{a2\}}\ \cdots\ s_k\ \mathsf{\}}) & \overset{def}{=} & (s_1\ (\#^{\sigma}_{x_1}(\mathsf{a1}))\ s_2\ (\#^{\sigma}_{x_2}(\mathsf{a2}))\ \cdots s_k) \\
 & & \qquad (x_i \in \{i, d, r\})
\end{array}
$$

Expressions:

$$
\begin{array}{rcl}
\#^{\sigma}_{x}(\mathsf{ident}) & \overset{def}{=} & \mathit{ident} \qquad (\mathit{ident}\ \text{a bound variable}) \\
\#^{\sigma}_{x}(\mathsf{\$a}) & \overset{def}{=} & a \\
\#^{\sigma}_{x}(\mathsf{lval}) & \overset{def}{=} & \#^{\sigma}_{lv}(\mathsf{lval})\ @_x\ \sigma \qquad (x \in \{i, d, r\}) \\
\#^{\sigma}_{x}(\mathsf{\backslash old(e)}) & \overset{def}{=} & \#^{\sigma_1}_{x}(\mathsf{e}) \\
\#^{\sigma}(\mathsf{e\ @\ l}) & \overset{def}{=} & \#^{\sigma_l}(\mathsf{e}) \qquad (\Gamma(l) = \sigma_l) \\
\#^{\sigma}_{x}(\mathsf{E1} + \mathsf{E2}) & \overset{def}{=} & \#^{\sigma}_{x}(\mathsf{E1}) + \#^{\sigma}_{x}(\mathsf{E2}) \qquad (x \in \{i, d\})
\end{array}
$$

Lvalues:

$$
\begin{array}{rcl}
\#^{\sigma}_{lv}(\mathsf{ident}) & \overset{def}{=} & \mathit{lookup\text{-}var}\ \Gamma\ \mathit{ident} \\
 & & \qquad (\mathit{ident}\ \text{a program variable}) \\
\#^{\sigma}_{lv}(\mathsf{lval[e]}) & \overset{def}{=} & (\#^{\sigma}_{lv}(\mathsf{lval}))_{ty}.[\#^{\sigma}_{i}(\mathsf{e})] \\
 & & \qquad (ty = \mathit{type\text{-}of}(\mathsf{lval[e]}))
\end{array}
$$

Figure 5.4: Rules for the translation from abstract to Isabelle syntax

expression types. The generated Isabelle term for a postcondition *Post* of an n-argument function will have the form

$$\lambda\ \Gamma\ vs\ \sigma_1\ (\Omega,\ \sigma_2).\ \mathit{case}\ vs\ \mathit{of}\ [v_1, \ldots, v_n] \Rightarrow \#^{\sigma_2}(\mathit{Post}) \tag{5.2}$$

The translation operation $\#^{\sigma}$ thus generates the *body* of the lambda term (5.2). This means we translate state predicates in the implicit context of an environment Γ, the pre-state σ_1 and post-state σ_2 as well as function argument values v_i and a result value Ω. The superscript σ indicates which state is *active*, i. e., being used for expression evaluation. The active state is only changed from the post-state to the pre-state or the state of a program label during the evaluation of \old and @ expressions. The translation via the $\#^{\sigma}$ resembles the side-effect free expression semantics on the quotation-free part of a specification term.

Fig. 5.4 shows representative translation rules. The logical constants and connectives are directly translated to their Isabelle equivalents. The keyword \ result becomes Ω, aggressively evaluated from *Val option* to one of the three domain value types, depending on the return type of the function. Quantifiers are directly interpreted as Isabelle/HOL quantifiers. We assume that the abstract syntax distinguishes between bound variables and identifiers for program variables. e @ l lets e be evaluated in the state at label l, which is looked up in the implicit environment Γ. Each pointer predicate directly corresponds to an Isabelle/HOL counterpart, e. g. pointer validity •(·.| ·) interprets **valid**. An auxiliary function *type-of*(·) provides the *RTT* type of an expression or lvalue. Quotations are output verbatim, except for anti-quotations, the translation of which is spliced into the quotation on the point of occurrence. The translation of expressions and lvalues behaves like the respective side-effect free semantic function, but outputs references to variables bound by quantifiers of the specification language (ident) as well as references to plain Isabelle variables ($a) by their name (*a*).

5.5.1 Type Checking

By allowing quotations of arbitrary Isabelle/HOL terms in specification expressions, Isabelle/HOL must be used to fulle type-check the latter. To allow for early detection of type errors, the front-end is used to check those parts of specification expressions outside of quotations, assuming that quotations have type **_Bool**. Type errors in anti-quotations are only discovered when the output of the front-end is parsed in by Isabelle/HOL. The return and argument types of Isabelle/HOL functions that are referred to syntactically explicitly (as in $f (...)) can be determined from their declaration (cf. Sec. 3.5.3) and the front-end uses this information during type-checking.

5.5.2 Modification Sets

The datatype for mvalues (cf. Sec. 3.5.1) equals that of lvalues, except that concrete array indices are replaced by intervals i : j. The denotation of an mvalue is a modification set and depends on the pre-state, e. g. to specify that *x is changed when passing a pointer x to a function. The front-end simply outputs an mvalue as an appropriate value of the datatype *MVal*, so that modification sets are evaluated in Isabelle/HOL as follows.

```
fun tl_mval :: "Env ⇒ State ⇒ MVal ⇒ Loc set"
where
  "tl_mval Γ σ (MId ident ty) = {lookup_var Γ ident}"
| "tl_mval Γ σ (MDeref mv ty) = (λl. l @l σ) ‘ (tl_mval Γ σ mv)"
| "tl_mval Γ σ (MArrayAcc mv indexpr ty) =
   (let ls = tl_mval Γ σ mv;
        rs = (λl. l @l σ) ‘ ls
    in (case indexpr of
         (MInterval a b) ⇒
           let i1 = sem_int_sef Γ a σ;
               i2 = sem_int_sef Γ b σ
           in
           {r_tyϑ.[nat i] | i r. i1 ≤ i ∧ i < i2 ∧ r ∈ rs}))
| "tl_mval Γ σ (MFieldSel mv f ty) =
    (let ls = tl_mval Γ σ mv
     in {r_tyϑ→f | r. r ∈ ls })"
```

Function *tl-mval* does not directly compute the set of locations, but only those locations at which modifiable objects lie in memory (called *start loca-*

tions in the following). The actual modification set can be computed from the start locations easily, by just expanding the computed set by all locations that comprise the modifiable object. For example, for an object declared as

```
struct ipt { int x; int y; } *p1;
```

for the mvalue `*p1` *tl-mval* will yield the start location l of the object that `p1` points to, which has to be expanded to $\{l, l_{ipt}\rightarrow"x", l_{ipt}\rightarrow"y"\}$ according to its type. This slightly roundabout definition allows us to define *tl-mval* in a structurally simple manner.

The modification set described by a simple identifier is the singleton set containing the location assigned to the identifier by Γ. Dereferencing a pointer means computing the set of locations denoted by the operand mv and then computing $l\ @_l\ \sigma$ for all locations l thus obtained. The modification set denoted by an array interval is the one which introduces sets of larger size; its start locations are all array accesses with indices between the lower and upper indices a and b on the locations obtained by evaluating the operand mvalue. For field selection we simply map the appropriate selector onto the set denoted by the operand.

Chapter 6

Hoare Logic and Verification Conditions

This chapter is concerned with the methodology and necessary infrastructure for proving programs correct. First, the notion of satisfaction between a program and its specification is formalised, akin to the notion of total correctness in Hoare logics. Modularity is achieved by verifying functions separately, and reusing already proven specifications. Proof rules allow for the derivation of verification conditions along the structure of the program syntax. The structure of these verification conditions is anslysed, which are supposed to be simplified interactively by the verifier, with the help of proof tactics provided for this purpose. Finally, the algorithmic structure of the main tactic for simplifying state changes in verification conditions, which we term read/update simplification, is described.

6.1 Specification Satisfaction

We need to define what it means for a program to satisfy its specification. At the lowest level, specification satisfaction is defined in terms of the program semantics. Given a state transformer p and a specification consisting of two state predicates P :: *unit SP* and Q :: *'b SP* and a modification set Λ, we say that p satisfies the specification (writing $\Lambda \models [P]\ p\ [Q]$) if p terminates for all states σ' in which P holds, yielding a result and post-state (a, σ) for which Q holds, and if the post-state differs from the pre-state only on locations in Λ. Formally we define

```
definition
  sat :: "Loc set ⇒ unitSP ⇒ 'b ST ⇒ 'b SP ⇒ bool"
where
  "(Λ ⊨ [P] p [Q]) =
   (∀σ'. P σ' ⟶ (∃ a σ. p σ' = Some (a, σ) ∧ Q a σ ∧ σ' ⊑Λ σ))"
```

This definition requires the termination of the state transformer and hence formalises a notion of total correctness. Since the semantics identifies all intermediate failures like dereferencing an invalid pointer with complete failure, $\Lambda \models [P]\ p\ [Q]$ implies that p only performs valid memory accesses. All particular notions of satisfaction such as those for expressions, statements, etc., will be based on this fundamental definition.

6.2 Modular Verification

One should be able to verify a single C function f when given its specification, the existence of all global variables referred to in f as well as the specifications of all functions called by f. This opens up the question about the concrete environment Γ (or set of environments) in which f should be verified. Recalling that the semantics of function calls is given by looking up the called function's semantics in the environment, and noting that during the verification of f some called function g might not have been implemented yet, we would like Γ to contain *some* semantics for g from which we only demand that it satisfies the —existing— specification of g.

Ultimately, we want to state that all functions are correct w. r. t. a single concrete environment containing the semantics of all actually implemented functions, of course. The lack of recursion and function pointers allows us to assume that we can assign a unique index $i \in \{1, \ldots, n\}$ to each program function such that $\forall f_j \in \mathit{calls}\ f_i.\ j < i$, where *calls* f_i yields all functions directly called by f_i. We can then subsequently enrich an initial environment Θ containing all global variables of the program and the specifications of all functions, but *no* semantics, by the semantics of individual functions, as follows:

$$\begin{aligned}
\Gamma_0 &= \Theta \\
\Gamma_1 &= \mathit{add\text{-}fun}\ \Gamma_0\ (\mathit{id\text{-}of}\ f_1)\ (\mathit{eval\text{-}spec}\ \Theta\ f_1,\ \mathit{Some}\ (\mathit{sem\text{-}fundef}\ \ \Gamma_0\ f_1)) \\
\Gamma_2 &= \mathit{add\text{-}fun}\ \Gamma_1\ (\mathit{id\text{-}of}\ f_2)\ (\mathit{eval\text{-}spec}\ \Theta\ f_2,\ \mathit{Some}\ (\mathit{sem\text{-}fundef}\ \ \Gamma_1\ f_2)) \\
&\vdots \\
\Gamma_n &= \mathit{add\text{-}fun}\ \Gamma_{n-1}\ (\mathit{id\text{-}of}\ f_n)\ (\mathit{eval\text{-}spec}\ \Theta\ f_n,\ \mathit{Some}\ (\mathit{sem\text{-}fundef}\ \ \Gamma_{n-1}\ f_n))
\end{aligned} \tag{6.1}$$

Γ_{i+1} equals Γ_i, except that it maps the function identifier f_{i+1} to its specification (evaluated in Θ, as specifications only depend on global variables, but not on function semantics or labels) and its semantics, which is evaluated in Γ_i. The latter ensures that the semantics are well-defined, since all functions called by f_{i+1} have their semantics assigned in Γ_i.

As argued above, we want to characterise the relevant properties of Γ_n before being able to construct it. The first step is to state that an environment maps a function identifier to a function satisfying the specification assigned to it:

```
definition
  fun_satisfies_spec :: "Env ⇒ Identifier ⇒ bool"
where
"fun_satisfies_spec Γ fid =
  (let ptypes = lookup_funargs Γ fid in
   let f_st   = lookup_funsem Γ fid in
   let Pre    = lookup_funspec_pre Γ fid in
   let Post   = lookup_funspec_post Γ fid in
   let locf   = lookup_funmod Γ fid in
     (∀args S1. args_types ptypes args ⟶
  ((locf args S1) ⊨ [(λS. S = S1 ∧ Pre args S)]
                      f_st args
                     [(λb S. Post args S1 (b, S))] )))"
```

All entities related to identifier *fid* are looked up in the environment; they are the parameter types *ptypes*, the semantics as a state transformer parameterised over argument values *f-st*, pre- and postcondition *Pre*, *Post*, and the modification set *locf* which also depends on arguments and a state. Function *fid satisfies its specification* if for all arguments *args* matching the types of the parameters (*args-types ptypes args*) the state transformer *f-st args* satisfies the displayed specification. We make use of a common trick [115] here, to turn *Post*, which is a state relation by nature, into a state predicate: higher-order logic allows us to quantify over states (here: *S1*); in the precondition we force it to be equal to the pre-state and use it in the post-condition as the according argument of *Post*. A similar definition exists for specification satisfaction by a function definition; here, all relevant entities are taken from the definition instead of being looked up in the environment, which is only used to evaluate the function's semantics:

```
  fundef_satisfies_spec :: "Env ⇒ Fundef ⇒ bool"
```

The second step is to characterise an environment as an *extension* of a given environment. An extension Γ of an environment Θ is one that maps all variables, labels and function specifications defined in Θ to the exact same values, but possibly has a larger domain:[1]

definition
```
  env_ext :: "Env ⇒ Env ⇒ bool"
```
where
```
  "env_ext Θ Γ =
    (varEnv Θ ⊆m varEnv Γ ∧ labEnv Θ ⊆m labEnv Γ ∧
     (Option.map fst ∘ funEnv Θ) ⊆m (Option.map fst ∘ funEnv Γ))"
```

A *correct extension* Γ of Θ w. r. t. a function definition f is an extension that additionally defines the semantics of all functions called by f such that they satisfy their specifications:

definition
```
  cenv_ext :: "Env ⇒ Fundef ⇒ Env ⇒ bool"
```
where
```
  "cenv_ext Θ f Γ =
    (env_ext Θ Γ ∧ (∀g ∈ calls f. fun_satisfies_spec Γ g))"
```

We write $\Theta \lhd_f \Gamma$ for *cenv-ext* Θ f Γ.

6.2.1 Modular Function Correctness

With these definitions we can state our notion of *modular function correctness*:

definition
```
  correct_function_modular :: "Env ⇒ Fundef ⇒ bool"
```
where
```
  "correct_function_modular Θ fd =
       (∀Γ. Θ ◁fd Γ ⟶  fundef_satisfies_spec Γ fd)"
```

A function definition *fd* is modularly correct w. r. t. an environment Θ (which will be Θ of Eq. (6.1) in concrete verifications) if we can prove that its definition satisfies its assigned specification in an environment which is a correct extension of Θ for *fd*. This proof can be done once and for all and is independent of any modifications in functions that are called by *fd*, as long as their specifications

[1] $\subseteq_m$ is the canonical partial function ordering satisfying $f \subseteq_m g$ iff $\forall x \in \mathit{dom}\ f.\ f\ x = g\ x$.

remain the same. As soon as Γ_n can be constructed, i. e., as soon as all program functions are defined, we can prove that $\Theta \lhd_{fd} \Gamma_n$ to obtain that *fd* satisfies its specification w. r. t. Γ_n.

6.3 Proof Rules

Approaches to program verification based on pre- and postconditions (e. g., [58, 96, 56, 13, 19, 43]) usually view the program as a predicate transformer and compute verification conditions (*VCs* for short) whose validity implies that the program satisfies its specification. This is done either by letting the program transform the precondition into a predicate that must imply the specified postcondition (*strongest postcondition* style VC generation), or by transforming the postcondition into a predicate that must be implied by the specified precondition (*weakest precondition* style). The verification conditions ideally do not make reference to any constructs of the programming language any more. They are derived through the application of appropriate proof rules of a program logic. As Hatcliff et al. [72] point out, it is more convenient to work in a program logic for a specific programming language than directly with the program semantics, because reusable proof principles can be encapsulated in the logic. More importantly, rules of the program logic are firstly syntax-driven and therefore suitable for an automatic application both in an interactive theorem prover and programmatic verification condition generators. Secondly, the program semantics do not take modularity considerations into account, which are of paramount importance for feasible verification.

6.3.1 Syntactic Notion of Satisfaction

The theoretically cleanest approach to defining a program logic in Isabelle-/HOL is to define an inductive set of derivable 'Hoare triples' (of the form $\Lambda, \Gamma \vdash [P]\ p\ [Q]$) where each introduction rule of the set represents a rule of the program logic. The validity of such a triple would be defined w. r. t. the semantic notion of satisfaction $\Lambda \models [P]\ \langle \textit{semantics of } p \textit{ in } \Gamma \rangle\ [Q]$, and soundness and (Cook) completeness of the program logic would be proven, asserting that the inductive set contains all and only the valid triples.

From a practical point of view, this strict separation of program logic and 'outer' logic in which the former is embedded is unnecessary: while we certainly require all derivation rules to be sound, the completeness property is only of theoretical interest, because all logically interesting derivation steps are done outside the program logic, by weakening or strengthening predicates of the outer logic, anyway. This is manifested in the usual Hoare rule for strengthening the precondition, where P, P' and Q are arbitrary Isabelle/HOL predicates:

$$\frac{P \longrightarrow P' \quad \Lambda, \Gamma \vdash [P']\ p\ [Q]}{\Lambda, \Gamma \vdash [P]\ p\ [Q]} \tag{6.2}$$

This observation motivates a shallow embedding of Hoare triples in which each triple is directly defined in terms of an appropriate 'semantic triple'. For example, the assertion that a syntactic integer expression satisfies its specification simply states that its semantics evaluated in Γ satisfies its specification:

definition

```
  sat_i' :: "Loc set ⇒ Env ⇒ unitSP ⇒ IntExpr ⇒ DomInt SP ⇒ bool"
where
  "sat_i' Λ Γ P ie Q = (Λ ⊨ [P] sem_int Γ ie [Q])"
```

Similarly we define the satisfaction for functions, given their parameters *params*, body *blk* and arguments *args*:

```
definition
  sat_params' :: "Loc set ⇒ Env ⇒ unitSP ⇒ Block ⇒
          Paramdecl list ⇒ Val list ⇒ (Val option) SP ⇒ bool"
where
  "sat_params' Λ Γ P blk params args Q =
     (Λ ⊨ [P] sem_paramdecls Γ blk params args [Q])"
```

We write $\Lambda, \Gamma \vdash_i [P]\ ie\ [Q]$ for *sat-i* $\Lambda\ \Gamma\ P\ ie\ Q$ and call Λ, Γ the *context* of the triple. All other syntactic categories have analogous definitions that are syntactically distinguished only by the respective subscript. ($\vdash_l$ for lvalues, $\vdash_r$ for pointer expressions, $\vdash_e$ for expressions, $\vdash_s$ for statements, etc.)

6.3.2 Proof Strategy

Our general strategy for proving the correctness of a function with definition *fd* w. r. t. the concrete initial environment Θ is to proceed in two phases. The modification set Λ, precondition P and postcondition Q, the function body and parameter list *params* can all be extracted from *fd*. We start from the correctness assertion *correct-function-modular fd* Θ and turn it into an appropriate triple in an environment that correctly extends Θ,

$$\bigwedge \Gamma\ S_1\ args.\ \llbracket\ \Theta \lhd_{fd} \Gamma\ \rrbracket \Longrightarrow \begin{array}{l} \Lambda, \Gamma \vdash_{params} [\lambda S.\ P\ \Gamma\ args\ S \wedge S = S_1] \\ \qquad \langle body\ params\ args \rangle \\ \qquad [\lambda b\ S.\ Q\ \Gamma\ args\ S_1\ (b, S)] \end{array},$$

on which an initial strengthening of the precondition P is performed, introducing two proof obligations

$$\begin{array}{ll} 1. & \bigwedge \Gamma\ S_1\ args.\ \llbracket\ \Theta \lhd_{fd} \Gamma\ \rrbracket \Longrightarrow \Lambda, \Gamma \vdash_{params} [?P'\ \Gamma\ S_1]\ \langle body\ params\ args \rangle \\ & \qquad\qquad\qquad\qquad\qquad\qquad [\lambda b\ S.\ Q\ \Gamma\ args\ S_1\ (b, S)] \\ 2. & \bigwedge \Gamma\ S_1\ args.\ P\ \Gamma\ args\ S_1 \longrightarrow\ ?P'\ \Gamma\ S_1\ S_1 \end{array} \tag{6.3}$$

where $?P'$ is a meta-variable that will be instantiated by applying the proof rules described in the subsequent sections with a concrete predicate R not containing meta-variables. We call the single remaining subgoal $P\ \Gamma\ args\ S_1 \longrightarrow R\ \Gamma\ S_1\ S_1$ the *initial verification condition* (iVC). At this point the second phase is entered, where the verifier interactively applies specialised tactics to discharge the proof obligation, as described in Sec. 6.5.

Our strategy is therefore similar to classical weakest precondition style verification condition generators (e. g., [96, 57]), because we also perform a backwards proof from Q to R along the program structure. However, it substantially differs in the structure of the generated R and the way that the implication from P is proven.

A Note on Rule Structure

We use unification to let VC computations for subexpressions communicate their results to the proof context in which they are applied. For example, $?P'$

will be instantiated with the result R of the VC computation in the first goal of Eq. (6.3), and unification makes this result appear in the second goal, which also contains $?P'$. To compute VCs fully automatically according to this strategy, all proof rules must exhibit a certain structure. The two conditions are that both the postcondition of a triple in the conclusion of a rule and the preconditions of triples in the premises must be plain meta-variables. This ensures that when we start with the goal state of Eq. (6.3) we can always apply the appropriate rules, and all subsequent goal states containing triples will also have a meta-variable in the precondition.

6.3.3 Lvalues and Expressions

A representative selection of rules for lvalues and integer expressions is depicted in Fig. 6.1. They are, as usual, best understood by reading them backwards. The rule for identifiers is trivial: since looking up an identifier i has no side-effects, we know that if *loc* is the location to which i gets mapped in Γ, then if the pre-state satisfies $Q\ loc$, the postcondition (which expects the returned location as an argument) will be $\lambda\ loc\ S.\ Q\ loc\ S$, or simply Q. To prove a triple for the dereferencing of a pointer expression e, we need to prove that evaluating e yields a location r that satisfies the postcondition Q and is valid for the type of the overall lvalue. Rule *PR-LArrayAcc* demonstrates how two triples are connected to account for the sequential evaluation inherent in the semantics; to prove an array access correct, we prove that the pointer expression a denoting the array satisfies a triple with intermediate postcondition $\lambda\ a\ S.\ R\ a\ S$, which is used as the precondition of the triple for the index expression i, in which a is now universally quantified. The postcondition of the latter triple obtains the value of i (named j) and requires Q to be satisfied for the location $a_{ty}.[nat\ j]$ and the array access at j to be valid. *PR-LFieldSel* follows the pattern of the previous two rules: we need to prove triples for the operands whose evaluation in the semantics may cause side-effects (in this case only the lvalue a whose field f is accessed), and ultimately reflect the additional effects that the overall lvalue/expression evaluation has (here: selecting a field) in the post-condition of the appropriate operand.

This scheme is also applied in the rules for binary operators, *PR-IntArith* and *PR-IntIntComp*. Functions *arith-op* and *comp-op* simply yield a binary function on domain values for their given arguments, e. g., $\lambda x\ y.\ x + y$ for *OpPlus* etc. Rule *PR-IntLVal* introduces the effect of reading the value at a location in the postcondition. It is crucial to let-bind this value to avoid a blow-up of the size of the initial verification condition: had we passed the read-term to Q directly instead (by writing $Q\ (lv\ @_i\ S)\ S$), $lv\ @_i\ S$ would possibly appear multiple times in the β-reduced postcondition. However, a term reading an intermediate lvalue in an intermediate state needs to be simplified during read/update simplification. Consequently, it is important to have it occur just once.

Rule *PR-IntLAnd* demonstrates that the injudicious combination of side-effects in expressions and non-strict operators like && leads to a duplication of verification conditions: to prove that e_1 && e_2 satisfies postcondition Q, we need to show that e_2 establishes Q when started in a state satisfying Qe, which is the postcondition established by e_1 in the case where it evaluates to *True*. Furthermore, e_1 must establish Q itself if it evaluates to *False*, since

Lvalues

```
  theorem PR_LId:
    "lookup_var Γ i = loc ⟹ Λ, Γ ⊢lv [(λS. Q loc S)] LId i t [Q]"

  theorem PR_LDeref:
    "⟦ (t_ϑ) = ty;
       Λ, Γ ⊢r [P] e [λr S. Q r S ∧ •(r_ty| S)] ⟧
     ⟹ Λ, Γ ⊢lv [P]LDeref e t[Q]"

  theorem PR_LArrayAcc:
    "⟦ (t_ϑ) = ty;
       ∀a. (Λ, Γ ⊢i [R a] i [(λj S. Q (a_ty.[nat j]) S ∧
                                        •[a_ty| j| S])]);
       Λ, Γ ⊢r [P] a [R] ⟧
     ⟹ Λ, Γ ⊢lv [P]LArrayAcc a i t[Q]"

  theorem PR_LFieldSel:
    "⟦ ((type_lv a)_ϑ) = ty;
     Λ, Γ ⊢lv [P] a [λb S. Q (b_ty→f) S] ⟧ ⟹
     Λ, Γ ⊢lv [P] LFieldSel a f t [Q]"
```

Integer expressions

```
  theorem PR_IntLVal:
    "⟦ Λ, Γ ⊢lv [P] l [(λlv S. let iv = (lv @i S) in Q iv S)] ⟧
     ⟹ Λ, Γ ⊢i [P] IntLVal l [Q]"

  theorem PR_IntArith:
    "⟦ arith_op ao = opf;
      ∀a. (Λ, Γ ⊢i [R a] t [λb S. Q (opf a b) S]);
       Λ, Γ ⊢i [P] s [R]⟧
     ⟹ Λ, Γ ⊢i [P] IntArith ao s t [Q]"

  theorem PR_IntIntComp:
    "⟦  comp_op co = cof;
        ∀a. (Λ, Γ ⊢i [R a] t [(λb S. Q (cof a b) S)]);
        Λ, Γ ⊢i [P] s [R] ⟧
    ⟹ Λ, Γ ⊢i [P] IntIntComp co s t [Q]"

  theorem PR_IntLAnd:
    "⟦ Λ, Γ ⊢b [Qe] e2 [λv S. Q (bool_to_int v) S];
       Λ, Γ ⊢b [P] e1 [λv S. (¬v ⟶ Q (bool_to_int v) S)
                ∧ (v ⟶ Qe S)] ⟧
     ⟹ Λ, Γ ⊢i [P] IntLAnd e1 e2 [Q]"
```

Figure 6.1: Proof rules for lvalues and expressions

non-strictness prevents the evaluation of e_2. Note that Q will generally not be a simple postcondition, but the (possibly large) computed initial verification condition for the expressions and statements following e_1 && e_2 in the overall function to be verified. Further note that the duplication is solely due to the possible side-effects that e_1 and e_2 might have; the result of the conjunction is *False* if e_1 evaluates to *False*, whether e_2 is evaluated or not. The duplication can be avoided if both e_1 and e_2 are side-effect free, since then their evaluation has no impact on the state and the result of a strict evaluation of the conjunction can directly be put into the precondition. Using *sef-sem-bool* of the side-effect free semantics (cf. Sec. 5.4) we can thus formulate a variant of the rule:

$$\frac{\forall\sigma.\ \mathit{syn\text{-}sef\text{-}expr}\ \Gamma\ (\mathit{IntLAnd}\ e_1\ e_2)\ \sigma}{\begin{array}{ll}\Lambda,\Gamma\vdash_i & [\lambda\sigma.\ Q\ ((\mathit{sef\text{-}sem\text{-}bool}\ \Gamma\ e_1\ \sigma)\wedge(\mathit{sef\text{-}sem\text{-}bool}\ \Gamma\ e_2\ \sigma))\ \sigma]\\ & \quad \mathit{IntLAnd}\ e_1\ e_2\\ & \quad [Q]\end{array}}$$

We observe that there is not much use in formulating the effect of side-effect free expression evaluation in terms of triples. Instead, the context in which the expression occurs should directly evaluate it via the side-effect free semantics. We have done this, e. g., in rule *PR-ConditionalStmt-sef* below.

For the sake of (relative) completeness, Fig. 6.2 displays further rules for several expression kinds. Rule *PR-IntRefDiff* is for pointer difference, another integer expression: the postcondition of the second operand r_2 requires that both references r_1 and r_2 point into the same array with initial element m with indices i and j, where m must denote an array large enough to make both i and j valid indices. The postcondition of the conclusion Q must then hold for the integer difference $i - j$.

The rules for pointer expressions are straightforward; Q *nulloc* must hold initially to make Q true after evaluating the NULL literal, where $\mathit{nulloc} = \mathit{Loc}\ \mathit{NullBase}\ 0$, i. e., the location of the null pointer in the memory model. The postcondition for the address-of operator (which expects a value of type *Loc*) and the triple for its operand (which evaluates its argument as an lvalue, hence a *Loc*, too) coincide. Pointer conversions to and from **void** $*$ (*PR-RefToVoid* and *PR-RefFromVoid*) are transparent, just as in the semantics.

Finally, there are rules for general expressions. *PR-IntExpr* simply delegates its work to a triple for integer expressions, where the postcondition lifts the obtained *DomInt* back into an expression value of type *Val*. If the expression is used in a boolean context (*PR-IntExpr-b*), the integer value *IntVal i* is evaluated as a boolean value. Expression lists, which appear as function arguments, have two rules for the empty and the non-empty expression list (*PR-ExprList-Nil* and *PR-ExprList-Cons*).

6.3.4 Function Calls and Statements

Function calls

The rules for function calls have a more elaborate structure than previous rules. We present the one for functions with an integer result, *PR-IntFunCall-3-modular*, in Fig. 6.3. In the semantics of calling a function *fid* within a function definition *fd* the arguments are evaluated, *fid* is looked up in the environment, and its semantics is applied to the arguments. For a modular verification, we

Integer expressions

```
theorem PR_IntRefDiff:
  "⟦ t'_ϑ = t;
     ∀a. (Λ, Γ ⊢r [R a] r2
           [(λb S. ∃m n i j. a = m_t.[i] ∧
                   b = m_t.[j] ∧ •[m_t| n| S] ∧
                   int i ≤ n + 1 ∧ int j ≤ n + 1 ∧
                   Q (int i - int j) S)]);
     Λ, Γ ⊢r [P] r1 [R]⟧
   ⟹ Λ, Γ ⊢i [P] IntRefDiff r1 r2 t' [Q]"
```

Pointer expressions

```
theorem PR_RefNull:
  "Λ, Γ ⊢r [Q nulloc] RefNull [Q]"

theorem PR_RefAddr:
  "⟦ Λ, Γ ⊢lv [P] lv [(λ r. Q r)] ⟧
   ⟹ Λ, Γ ⊢r [P] RefAddr lv [Q]"

theorem PR_RefToVoid:
  " Λ, Γ ⊢r [P] re [R] ⟹
    Λ, Γ ⊢r [P] RefToVoid re [R]"

theorem PR_RefFromVoid:
  " Λ, Γ ⊢r [P] re [R] ⟹
    Λ, Γ ⊢r [P] RefFromVoid re t [R]"
```

Expressions

```
theorem PR_IntExpr:
  "Λ, Γ ⊢i [P] e [λi. Q (IntVal i)]
  ⟹ Λ, Γ ⊢e [P] IntExpr e [Q]"

theorem PR_IntExpr_b:
 "Λ, Γ ⊢i [P] e [λi. Q (is_true_val (IntVal i))]
  ⟹ Λ, Γ ⊢b [P] IntExpr e [Q]"

theorem PR_ExprList_Nil:
  "Λ, Γ ⊢es [P []] ExprList_Nil [P]"

theorem PR_ExprList_Cons:
  "⟦ ∀b. (Λ, Γ ⊢es [Q b] es [λbs. R (b#bs)]);
     Λ, Γ ⊢e [P] e [Q] ⟧
   ⟹ Λ, Γ ⊢es [P] ExprList_Cons e es [R]"
```

Figure 6.2: Proof rules for expressions

only want to depend on *specification* information. To ensure that Γ contains a proper semantics for *fid* satisfying its specification as given by the initial environment Θ, we require that $\Theta \lhd_{fd} \Gamma$. Since *fid* is an arbitrary identifier as far as the function call rule is concerned, we must formally demand that it be mapped by Θ (*env-maps-funid fid* Θ). Furthermore, *fd* must actually call *fid* ($\mathit{fid} \in \mathit{calls}\ \mathit{fd}$)[2].

To ensure a postcondition Q after a function call, then, four conditions need to be true for the state S after the evaluation of the function arguments (*vs*):

1. The called function's precondition must hold

$$\mathit{lookup\text{-}funspec\text{-}pre}\ \Theta\ \mathit{fid}\ \mathit{vs}\ S,$$

2. the types of the values obtained from evaluating the arguments must match the types of the formal parameters

$$\mathit{args\text{-}types}\ (\mathit{lookup\text{-}funargs}\ \Theta\ \mathit{fid})\ \mathit{vs},$$

3. the modification set of the called function, interpreted in S, must be a subset of the modification set Λ of the conclusion

$$\mathit{lookup\text{-}funmod}\ \Theta\ \mathit{fid}\ \mathit{vs}\ S \subseteq \Lambda,$$

4. the state T after the function call must satisfy Q $(\mathit{valToInt}\ (\mathit{the}\ a))\ T$. However, we cannot precisely characterise T; we only know it satisfies the called function's postcondition

$$\mathit{lookup\text{-}funspec\text{-}post}\ \Theta\ \mathit{fid}\ \mathit{vs}\ S\ (a, T)$$

and, importantly, that it equals S on all locations but those in the modification set of *fid* (denoted by X hereafter):

$$S \sqsubseteq_{\mathit{lookup\text{-}funmod}\ \Theta\ \mathit{fid}\ \mathit{vs}\ S} T$$

Note how a *local weakening* from *fid*'s postcondition to Q is achieved here w. r. t. T, *within* the postcondition of a triple. Strictly speaking, *fid*'s postcondition —which locally specifies *fid*'s effects— cannot imply Q, which generally will incorporate a larger context, e. g. reference variables not known to the former. However, we can strengthen the premiss of the implication by $S \sqsubseteq_X T$, which is in fact a strong assertion: all information about S that does not depend on the locations in X also holds for T. Since S is the state obtained by evaluating the function arguments in some previous state S' satisfying P, we obtain enough contextual information about S to prove the implication (only if the conclusion of the rule, $\Lambda, \Gamma \vdash_i [P]\ \mathit{IntFunCall}\ \mathit{fid}\ \mathit{es}\ [Q]$, is a valid triple, of course).

This formulation is not necessary to prove a function call rule correct at all; ignoring side-effects in the function arguments, a much simpler rule would identify P and Q in the conclusion with the pre- and postcondition of the called function:

$$\Lambda, \Gamma \vdash_i [\langle \mathit{pre}(\mathit{fid})\ \mathit{in}\ \Gamma\rangle]\ \mathit{IntFunCall}\ \mathit{fid}\ \mathit{es}\ [\langle \mathit{post}(\mathit{fid})\ \mathit{in}\ \Gamma\rangle]$$

[2] While it is structurally obvious that all functions *fid* in the body of *fd* are called by *fd*, the mere rule has no notion of being only applied to subterms of *fd*, which enforces an explicit premiss.

This rule, however, could only be ‘adapted’ for actual use by applying a global weakening as in Eq. (6.2). Since the pre- and postcondition are local to the function they specify, such a weakening will not succeed. In brief, our notion of framing through modification sets requires a local weakening inside preconditions or postconditions, not a global one.

Statements

The rules for the empty statement, the sequencing statement and expression statements in Fig. 6.3 are entirely standard. Since our pre- and postconditions are shallowly embedded, we cannot use substitution to reflect assignments in predicates. Instead, we explicitly modify the program state. Rule *PR-AssignStmt* shows this: to prove an assignment with postcondition Q, we evaluate the lvalue we assign to (in the second premiss), demanding it is a modifiable location ($l \in \Lambda$). We then evaluate the expression t (in the first premiss); the updated state is bound to an auxiliary variable S', which is passed to Q. This is equivalent to substitution: we create the predicate stating that updating the state at lv with t yields a state satisfying Q. The state predicate R is both the precondition of the first premiss and the postcondition of the second, thus again logically connecting Q and P in the conclusion. Conditional statements with side-effecting conditions are handled by rule *PR-ConditionalStmt*. Both branches s_1 and s_2 need to establish the postcondition R, but may assume different preconditions Qt and Qe. After evaluation of the condition c, either of these must hold, depending on the value b of the condition. Like rule *PR-IntLAnd*, this rule leads to a duplication of the postcondition R.

Specification statements Sec. 3.5.2 discussed the use of statement specifications to avoid this duplication and Sec. 5.2.2 presented the translation of a @join annotation to the *SpecStmt* constructor. The according proof rule is *PR-SpecStmt*. It is an example in which the precondition of the conclusion is not merely a variable. We call μ the modification set and φ the translation of the @join predicate given by the statement specification for statement s. The goal is to bound the duplications of the postcondition that are introduced by s (e. g. due to conditional statements in s) to φ (which is small), while retaining a single reference to the overall postcondition R (which may be large). We therefore introduce the premiss that s establishes φ when started in a state satisfying a precondition P, only modifying locations in μ. We effectively start a new verification condition generation w. r. t. s, starting from φ. The precondition for the *SpecStmt* structurally resembles the local weakening of the function call rule: P must hold, μ must be a subset of the specified modification set Λ, and the state T obtained from executing s, which we again cannot characterise precisely but for which $S \sqsubseteq_\mu T$ holds and which satisfies φ, must make the postcondition R true. If we read the rule operationally from the viewpoint of backwards verification condition generation, the key property of this rule is that R is isolated from s such that P's size only depends on the structure of s and φ.

Loops The rule for loops with possible side-effects in conditions is *PR-While-Stmt-modlist*, shown in Fig. 6.4. Its rather cryptic appearance is mostly due to the fact that all variables depending on quantified variables need to make this dependency explicit. E. g., the precondition of the loop body, J, must be

Function Calls

```
theorem PR_IntFunCall_3_modular:
"⟦ Θ ◁fd Γ;
   fid ∈ calls fd;
   env_maps_funid fid Θ;
   Λ, Γ ⊢es [P] es
    [λvs S. lookup_funspec_pre Θ fid vs S ∧
            args_types (lookup_funargs Θ fid) vs ∧
           (∀ (a::Val option) T. S ⊑lookup_funmod Θ fid vs S T
           ⟶ lookup_funspec_post Θ fid vs S (a, T)
           ⟶ Q (valToInt (the a)) T) ∧
               lookup_funmod Θ fid vs S ⊆ Λ ] ⟧
 ⟹ Λ, Γ ⊢i [P] IntFunCall fid es [Q] "
```

Statements

```
theorem PR_EmptyStmt:
  "Λ, Γ ⊢s [P] EmptyStmt [P]"

theorem PR_SeqStmt:
  "⟦ Λ, Γ ⊢s [Q] s2 [R];
    Λ, Γ ⊢s [P] s1 [Q] ⟧
   ⟹ Λ, Γ ⊢s [P] SeqStmt s1 s2 [R]"

theorem PR_AssignStmt:
  "⟦ ∀l. (Λ, Γ ⊢e [R l] t [λa S. ∀S'. S(l ::= a) = S' ⟶ Q S']);

     Λ, Γ ⊢lv [P] lv [λl S. R l S ∧ l ∈ Λ] ⟧
   ⟹ Λ, Γ ⊢s [P] AssignStmt lv t[Q]"

theorem PR_ConditionalStmt:
  "⟦ Λ, Γ ⊢s [Qt] s1 [R];
     Λ, Γ ⊢s [Qe] s2 [R];
     Λ, Γ ⊢b [P] c [λb S. (b ⟶ Qt S) ∧ (¬b ⟶ Qe S)] ⟧
   ⟹ Λ, Γ ⊢s [P] ConditionalStmt c s1 s2 [R]"

theorem PR_SpecStmt:
" ⟦ modified_locs Γ mlist = ModlF;
    spec Γ = φ;
    ∀μ σ. ModlF σ = μ ⟶
      (μ, Γ ⊢s [P μ σ] s [(λS. φ σ ((), S))]) ⟧
  ⟹ Λ, Γ ⊢s [(λS. ∀μ. ModlF S = μ ⟶
                        (P μ S S ∧ μ ⊆ Λ ∧
                         (∀T. S ⊑μ T ⟶
                              φ S ((), T) ⟶ R T)))]
              SpecStmt s (StmtAnno spec mlist) [R]"

theorem PR_ExprStmt:
  "Λ, Γ ⊢e [P] e [(λv. Q)]
  ⟹ Λ, Γ ⊢s [P] ExprStmt e [Q]"
```

Figure 6.3: Proof rules for statements

written as $J\ \mu\ N$ because it depends both on the quantified location set μ and the quantified termination counter N. We omit these technical arguments in the following and call φ the annotated invariant, μ its associated modification set, and Δ the measure function of type *State* ⇒ *nat*.

The precondition of the loop triple in the conclusion requires that the invariant φ holds; that μ is a subset of the set of modifiable locations Λ in the context; and that in each state T with $S \sqsubseteq_\mu T$ we may infer a predicate K from the invariant. (This, again, is a local weakening.) We want to show that the postcondition F holds after execution of the while statement. This holds given two premises. The first premiss states that an arbitrary run of the body c in a state satisfying J re-establishes the invariant φ. The second premiss states that after evaluation of the condition b under the precondition K we either obtain F directly —if b evaluates to *False*—, or we obtain a state satisfying the intermediate predicate J —if b evaluates to *True*. Both premises are formulated in the context of the annotated modification set μ, instead of the context Λ of the loop itself, ensuring that the loop condition and body only modify locations as annotated in the loop specification.

Termination of the loop is also ensured, employing the annotated variant Δ. Intuitively, the rule encodes the requirement that the variant, mapping program states to natural numbers, strictly decreases in each iteration. To achieve this, we introduce a universally quantified natural number N, of which we require that $\Delta\ T < N$ in the postcondition of the loop body. A typical formulation of loop termination might require $\Delta\ S = N$ in the precondition of the loop body to enforce a decrease of the measure in one iteration. This however violates the requirement of Sec. 6.3.2 that all preconditions in premises are plain meta-variables. Therefore, no additional requirement is placed on N in the precondition K, but K merely becomes a function over N. By doing so, the overall premiss states that whenever a state satisfies K for an arbitrary number N, then after execution of the loop condition and body it will end in a state in which the measure yields a number strictly smaller than N. In the conclusion we connect the actual measure to N by applying K to $\Delta\ T$, where T now stands for an arbitrary state at the beginning of a loop iteration.

Side-effect freeness

Rules *PR-ConditionalStmt-sef* and *PR-WhileStmt-sef* can be used when the evaluation of the respective condition is guaranteed to be side-effect free, which is the most common case. We look at the former rule here. The triple for the evaluation of the condition c is replaced by a direct call to the side-effect free semantics: for brevity we let *EP* denote the predicate determining whether c is side-effect free and *Sem* denote its value under these semantics. Under the premiss that the two branches s_1 and s_2 establish the postcondition R under the preconditions Qt and Qe, respectively, we know that the conditional statement will establish R when executed in a state S that ensures *EP* and which either satisfies Qt or Qe, depending on the boolean value of *Sem* S. Since *EP* is integrated into the precondition it is *not* required that c be side-effect free under all circumstances: using array accesses and dereferencing pointers in c is fine because *EP* will generate the necessary validity constraints in state S. The only case in which the rule cannot be used is when function calls occur in c, because *syn-sef-expr* over-approximates function calls as always having side-effects.

Statements

```
theorem PR_WhileStmt_modlist:
"⟦ modified_locs Γ mlist = μ';
   invar Γ () = φ;
   var Γ = Δ;
   ∀μ S N. μ' S = μ ⟶
    (μ, Γ ⊢s [J μ N] c [(λT. φ T ∧ Δ T < N)]) ∧
    (μ, Γ ⊢b [K μ N] b [(λb T. (b ⟶ J μ N T) ∧ (¬b ⟶ F T))])⟧

  ⟹ Λ, Γ ⊢s [(λS. ∀μ. μ' S = μ ⟶
                 (φ S ∧ μ ⊆ Λ ∧
                 (∀T. S ⊑μ T ⟶
                     φ T ⟶ K μ (Δ T) T)))]
      WhileStmt b c (Some (LoopAnno invar (Some mlist) (Some var)))
          [F]"


theorem PR_Label:
"⟦ ∀S'. (Λ, add_label Γ l S' ⊢s [P S'] p [Q]) ⟧
 ⟹ Λ, Γ ⊢s [(λS. P S S)] LabeledStmt l p [Q]"

theorem PR_ConditionalStmt_sef:
  "⟦ ⋀S. syn_sef_expr Γ c S = EP S;
     ⋀S. sef_sem_expr Γ c S = Sem S;
     Λ, Γ ⊢s [Qt] s1 [R];  Λ, Γ ⊢s [Qe] s2 [R] ⟧
   ⟹ Λ, Γ ⊢s [(λS. EP S ∧
      (is_true_val (Sem S) ⟶ Qt S) ∧
    (¬ is_true_val (Sem S) ⟶ Qe S))] ConditionalStmt c s1 s2 [R]"

theorem PR_WhileStmt_sef:
 "⟦ modified_locs Γ mlist = M;
   invar Γ () = Φ;
   var Γ = Δ;
   ⋀S. syn_sef_expr Γ b S = EP S;
   ⋀S. sef_sem_expr Γ b S = Sem S;
   ⋀ S N. (M S, Γ ⊢s [J (M S) N] c [(λT. Φ T ∧ Δ T < N)]) ⟧
  ⟹ Λ, Γ ⊢s [(λS. ∀M'. M S = M' ⟶
            (Φ S ∧ M' ⊆ Λ ∧
            (∀T. S ⊑M' T ⟶
             Φ T ⟶ EP T ∧
                (is_true_val (Sem T) ⟶ J M' (Δ T) T) ∧
              (¬ is_true_val (Sem T) ⟶ F T))))]
      WhileStmt b c (Some (LoopAnno invar (Some mlist) (Some var)))
             [F]"
```

Figure 6.4: More proof rules for statements

Weakening

```
  theorem PR_wk_pre_s[rule_format]:
    "⟦ Λ, Γ ⊢s [P'] c [Q]; ∀S. P S ⟶ P' S ⟧
     ⟹ Λ, Γ ⊢s [P] c [Q]"
```

Declarations

```
  theorem PR_Params_Nil:
  "Λ, Γ ⊢block [P] blk [Q] ⟹  Λ, Γ ⊢params [P] blk [] [] [Q]"

  theorem PR_Params_Cons:
  "⟦ ∀ n. (Λ ∪ ⇑_L Local p n, (add_var Γ p (ν·p_n))
      ⊢params [P n]
                blk ps z
               [(λb S. ∀S'. S⊖(ν·p_n) = S' ⟶ Q b S')]) ⟧
    ⟹ Λ, Γ ⊢params [(λS. ∀ n. (ν·p_n) ∉_S S ⟶
                          (∀S'. S⊗(ν·p_n, a1, ty_ϑ) = S' ⟶
                                P n S'))]
                       blk ((Paramdecl p ty)#ps) (a1#z)
                     [Q]"
```

Modular function correctness

```
  theorem PR_correct_function_modular:
  " ∀Γ. Θ ◁_f Γ ⟶
    (∀args S1. args_types (funParams (funHeader f)) args ⟶
           (sem_funmod Γ (funHeader f) args S1, Γ ⊢params
        [(λS. S = S1 ∧ funPrecondition Γ (funHeader f) args S)]
              (funBlock f) (funParams (funHeader f)) args
        [(λb T. funPostcondition Γ (funHeader f) args S1 (b, T))]))
    ⟹ correct_function_modular Θ f"
```

Figure 6.5: Proof rules for weakening, declarations and modular function correctness

6.3.5 Declarations and Weakening

The final set of rules —a selection of which are displayed in Fig. 6.5— are concerned with global predicate weakening, declarations of function parameters and of local variables, and finally the translation of the initial proof obligation *correct-function-modular* Θ *fid* into a triple. Rule *PR-wk-pre-s* formalises the strengthening of a precondition, which becomes a weakening when read backwards: to prove a triple under the precondition P, it suffices to prove it under the *weaker* precondition P'. We therefore subsume both the strengthening of a precondition and the weakening of a postcondition under the general term *weakening.* Such rules exist for all kinds of triples, but during verification condition generation only a single initial weakening is performed, due to the specific structure of the proof rules which have the necessary (local) weakening already built in.

Rule *PR-Params-Cons* describes the effect on predicates of introducing a locally scoped variable, in this case a function parameter: *blk* denotes the function body, *Paramdecl p ty* is a function parameter, a_1 the according argument value, and *ps* and z are the remaining parameters and arguments. To prove the triple with postcondition Q means to prove that Q is established after allocating a location for p, continuing with *ps*, evaluating the body *blk* in the extended state and deallocating all local variables afterwards. Therefore, the premiss requires that given a precondition P, *blk*, *ps* and z establish the postcondition in which Q holds after deallocating the location $\nu \cdot p_n$. This triple must be valid for the modification set Λ extended by the allocated variable $\nu \cdot p_n$ (and all its offsets), since local variables may always be modified by the functions defining them. The precondition of the conclusion must anticipate the allocation for parameter p. It consequently assumes a fresh location $\nu \cdot p_n$ for its state S and requires that the state S' which extends S by $\nu \cdot p_n$, assigning it the value a_1, satisfies P. This rule is correct because the state model enforces that all states have a finite domain and that we can always allocate another location. Otherwise, the precondition in the conclusion would explicitly have to ensure these properties, eventually leading to additional verification conditions to be proven by the verifier.

Rule *PR-correct-function-modular* allows to conclude that a function definition f is modularly correct w. r. t. an environment Θ by proving a $\vdash_{params}$ triple whose structure we have described in Sec. 6.3.2 and which is constructed from the terms contained in f.

6.4 Structure of the Initial Verification Condition

Approaches to program verification that generate verification conditions and pass them to an automatic prover (e. g., [14, 96, 56, 58]) optimise the structure of the generated VC for the prover, to either enable automatic proofs at all or to speed up the proof process. The common method, as described by Leino [96], is to transform the input program into a minimalist intermediate language of guarded commands. The translated program is further massaged within this language to take on a single static assignment form (SSA, [50]), where each program variable is assigned to exactly once. Finally, a weakest precondition style verification condition generator is applied to the SSA program. This sequence of transformations leads to VCs that are hardly comprehensible by a human verifier. This is not a problem in those cases where the VC is successfully proven by the associated automatic prover. But if the proof fails, tools such as Spec# [14] can merely point the verifier at a source code location which possibly caused or at least contributed to the error. A logical analysis of the error w. r. t. the VCs is generally not feasible.

In our case, it is essential that a verifier understands the structure of the initial VC, as it is his task to prove it. As pointed out earlier, the iVC is an implication $P \longrightarrow R$, where P is the function's precondition and R generated by application of the proof rules. An analysis of the rules reveals that R is a term constructed according to the following simple grammar:

$$\begin{array}{rcll} R & ::= & l \notin_S S \longrightarrow \forall S'.\ S\oplus(l,t) = S' \longrightarrow R_1 & \text{(ext)} \\ & | & \forall S'.\ S(l ::= e) = S' \longrightarrow R_1 & \text{(upd)} \\ & | & \mathbf{let}\ v = l \mathbin{@_x} S\ \mathbf{in}\ R_1 & \text{(read)} \\ & | & R_1 \wedge \bullet(l_t|\ S) & \text{(deref)} \\ & | & R_1 \wedge \bullet[l_t|\ n|\ S] & \text{(array)} \\ & | & (e \longrightarrow R_1) \wedge (\neg e \longrightarrow R_2) & \text{(cond)} \\ & | & R_1 \wedge (\mu \subseteq \Lambda) \wedge (\forall T.\ S \sqsubseteq_\mu T \longrightarrow R_2 \longrightarrow R_3) & \text{(wk)} \\ & | & R_1 \wedge \Delta < \Delta' & \text{(var)} \\ & | & \Phi \quad (\textit{annotated specification expression}) & \text{(spec)} \end{array} \tag{6.4}$$

In this grammar, l denotes terms built over location variables and the access operations $l_t{\to}f$ and $l_t.[n]$; S, S', and T are program state variables; e denotes arithmetic expressions built over let-bound variables, such as $(v_1+1)*v_2$; μ and Λ denote modification sets; Δ and Δ' denote evaluations of annotated variants in different program states; Φ represents specification expressions occurring in the function to be verified, such as invariants and pre-/postconditions of called functions; and the R_i denote subexpressions constructed recursively.

The first three grammar rules introduce new variables. Each extension or update on the state as well as each reference to an lvalue in an expression in a function results in an occurrence of a term of these shapes. Thanks to these bindings we avoid a combinatorial explosion of the size of the VC similar to [57], because no compound term can appear more than once. Terms described by (deref) and (array) are introduced by pointer dereferencing and array access, while (cond) describes the split introduced by if-statements. Interestingly, the shape of terms generated by function calls, specification statements and while loops is the same; it is the one described by (wk). In each case, we can understand R_1 as the condition that must hold in the pre-state of the respective statement or expression (e. g., a called function's precondition), R_2 as the condition ensured by its execution (e. g., the corresponding postcondition), and R_3 as the condition to be established afterwards. Finally, (var) describes terms introduced by termination requirements of loops. Both Δ and Δ' are evaluations of variant expressions in program states.

6.4.1 Example

Consider the following simple function which we use to illustrate the concrete shape of an iVC:

```
int set(int *a, int b) {
  if (a != NULL) {
    *a = b;
  }
  return;
}
```

We omitted the specification here and assume that after the translation of this code by the front-end and after the computation of the iVC P, Q, Λ_0 are the precondition, postcondition, and modification set, respectively. The iVC is displayed in Fig. 6.6. Ignoring the syntactic noise we clearly see that it encodes all possible execution paths of the program, together with all necessary validity

$$
\begin{array}{l}
[\![P\ S_1;\ \Lambda = \Lambda_0 \cup \Uparrow_L \nu\cdot\text{"}a\text{"}_n \cup \Uparrow_L \nu\cdot\text{"}b\text{"}_{n'}]\!] \Longrightarrow \\
\nu\cdot\text{"}a\text{"}_n \notin_S S_1 \longrightarrow \\
\quad (\forall S_2.\ S_1{\otimes}(\nu\cdot\text{"}a\text{"}_n, v_a, \mathit{ptr}) = S_2 \longrightarrow \\
\quad\quad \nu\cdot\text{"}b\text{"}_{n'} \notin_S S_2 \longrightarrow \\
\quad\quad\quad (\forall S_3.\ S_2{\otimes}(\nu\cdot\text{"}b\text{"}_{n'}, v_b, \mathit{int}) = S_3 \longrightarrow \\
(\textbf{let}\ c = \nu\cdot\text{"}a\text{"}_n \mathbin{@_l} S_3\ \textbf{in} \\
\quad (c \neq \mathit{nulloc} \longrightarrow \\
\quad\quad (\textbf{let}\ a = \nu\cdot\text{"}a\text{"}_n \mathbin{@_l} S_3\ \textbf{in} \\
\quad\quad\quad ((\textbf{let}\ b = \nu\cdot\text{"}b\text{"}_{n'} \mathbin{@_i} S_3\ \textbf{in} \\
\quad\quad\quad\quad\quad \forall S_4.\ S_3(a ::= \mathit{IntVal}\ b) = S_4 \longrightarrow \\
\quad\quad\quad\quad\quad\quad (\forall S_5.\ S_4{\ominus}\nu\cdot\text{"}b\text{"}_{n'} = S_5 \longrightarrow \\
\quad\quad\quad\quad\quad\quad\quad (\forall S_6.\ S_5{\ominus}\nu\cdot\text{"}a\text{"}_n = S_6 \longrightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad Q\ S_1\ (\mathit{None}, S_6)))) \wedge a \in \Lambda) \\
\quad\quad\quad \wedge \bullet\ (a_{\mathit{int}}|\ S_3))) \wedge \\
\quad (c = \mathit{nulloc} \longrightarrow \\
\quad\quad (\forall S'_4.\ S_3{\ominus}\nu\cdot\text{"}b\text{"}_{n'} = S'_4 \longrightarrow \\
\quad\quad\quad (\forall S'_5.\ S'_4{\ominus}\nu\cdot\text{"}a\text{"}_n = S'_5 \longrightarrow \\
\quad\quad\quad\quad Q\ S_1\ (\mathit{None}, S'_5)))))))
\end{array}
$$

Figure 6.6: Initial verification condition of the example program

conditions ($\bullet(a_{int}|\ S_3)$) and checks for allowed modifications ($a \in \Lambda$). In this case, there are only two paths, both of which finally require that Q holds. They initially allocate fresh locations for the parameters and assign them the input values (v_a, v_b). The first path then assumes $c \neq \mathit{nulloc}$, reads the values of a and b and introduces a successor state S_4 reflecting the assignment. The second path assumes $c = \mathit{nulloc}$ and immediately deallocates the local variables, subsequently requiring Q. We argue that a verifier can still maintain the connection between the concrete source code and the generated iVC. This holds true also for larger programs for two reasons. First, all updated intermediate states and all read lvalues are bound to variables and hence the substitutions inherent in classical formulations of Hoare rules or weakest precondition equations are *delayed*. The control flow defined by the source code can be mapped directly to the structure of the iVC. Second, the grammar of Eq. (6.4) can be used as a guide through the structure of the formula.

Another example including a concrete specification using representation functions is given in the Appendix in Sec. B.1.

6.4.2 Structural Simplification

We have said that the structure of the iVC encodes all possible paths through a function. More precisely, it encodes all *static paths* defined by the syntactic structure of the function, encoding loops and function calls as abstract state transitions as described by grammar rule (wk). While every conditional introduces a branch in the iVC (grammar rule (cond)), semantically certain (combinations of) branches might never be executed. Static paths which can never be executed are called *infeasible paths* in the literature. The following code snippet is an easy example with an infeasible then-branch:

```
x = 0;
if (x) { y = 1; }
```

It turned out useful during practical verification to eliminate two simple cases of infeasible paths as an initial simplification step. These paths were introduced by restrictions set up in the MISRA C programming guidelines.

Failure Propagation

The first kind of infeasible path is introduced by the particular failure propagation that was adopted in the SAMS code. Due to the MISRA requirement that each function shall have but one exit point and due to the absence of an exception mechanism in C, recognised failures during a computation have to be manually propagated to the end of the function body. This can in principle be done without the introduction of infeasible paths by an intricate nesting of conditional statements. However, such an approach makes the code hard to read. Our solution was to conceptually divide the function body into logical units and to protect each unit by an if-statement checking for a function-local failure status. Whenever a unit detects a failure, it sets the status accordingly, and thereby prevents all subsequent units from being executed. In code this looks as follows:

```
if (OK) { /* statements possibly modifying OK */ }
if (OK) { ... }
/* etc. */
```

This approach introduces infeasible paths: it is impossible to enter the then-branch of the second if-statement if OK was false before the first one. However, the iVC will initially contain this path:

$$\begin{array}{l}
\textbf{let } \mathit{ok} = \nu\cdot\text{"}OK\text{"}_n \,@_i\, S \textbf{ in} \\
\quad (\mathit{ok} \neq 0 \longrightarrow P_1) \wedge \\
\quad (\mathit{ok} = 0 \longrightarrow \\
\quad\quad (\textbf{let } \mathit{ok'} = \nu\cdot\text{"}OK\text{"}_n \,@_i\, S \textbf{ in} \\
\quad\quad\quad (\mathit{ok'} \neq 0 \longrightarrow P_2) \wedge \\
\quad\quad\quad (\mathit{ok'} = 0 \longrightarrow P_3)))
\end{array} \tag{6.5}$$

The condition $\mathit{ok'} \neq 0$ can obviously never be true given $\mathit{ok} = 0$. We have developed simplification tactics that erase infeasible paths such as this one from the iVC, rewriting it to the equivalent term

$$\begin{array}{l}
\textbf{let } \mathit{ok} = \nu\cdot\text{"}OK\text{"}_n \,@_i\, S \textbf{ in} \\
\quad (\mathit{ok} \neq 0 \longrightarrow P_1) \wedge \\
\quad (\mathit{ok} = 0 \longrightarrow P_3)
\end{array} \tag{6.6}$$

Defensive Programming

Defensive programming is required for safety-related software. It includes the cautious treatment of a function's input parameters by performing sanity checks on their values. This can make the iVC larger than necessary, when a function's precondition already precludes certain input vectors, which are nonetheless checked within the function to conform to defensive programming. For example, the computation of a square root might be specified and (partially) defined as follows:

```
/*@
  @requires x >= 0
  @modifies errno
  @ensures $abs(\result * \result - x) <= $eps
  @*/
double sqrt(double x) {
  if (x >= 0) { /* Compute square root */ }
  else { /* Some error notification */ }
}
```

The generally sensible check in the code that x is non-negative will always yield true whenever the function is used in a program in which all functions calling sqrt are formally verified. Moreover, during verification of sqrt we assume the precondition x >= 0 in any case. Our simplification tactics can perform a similar reduction of the iVC in this case like from Eq. (6.5) to Eq. (6.6) by eliminating the else-branch.

6.5 Tactics for Simplifying Verification Conditions

In this section we explain how the properties of the memory model presented in Chapter 4 are put to use during the reduction of the iVC to either program safety VCs or domain-related property VCs and present the tactics developed for this purpose. We stop at the point where all safety VCs have been proven and all effects of the program have been reflected in the domain-related VCs. The latter condition is satisfied if all remaining proof goals either do not refer to *any* program state except the initial one any more, or if all references to a program state are such that the relation of the state to other program states is irrelevant. How these remaining proof goals can be closed strongly depends on the concrete domain and its associated theorems. We provide examples of concrete verifications done in the SAMS project in the next chapter.

6.5.1 Stepping Through the iVC

We view the simplified iVC as a tree described by the grammar rules of Eq. (6.4). The overall strategy is that the verifier can employ tactics to traverse this tree from the root to the nodes by transforming the proof state appropriately, as well as to prove or at least simplify these nodes. A single tactic called `pr_step` realises the step-wise traversal of the tree. We define its mode of operation w. r. t. the grammar rules and the effect on the proof state:

(ext) Assume the freshness condition $l \notin_S S$ as well as the state equality $S \oplus (l, t) = S'$ (introducing a fresh variable S') and yield the new proof goal R_1.

(upd) Assume the state equality $S(l ::= e) = S'$ for a freshly introduced variable S' and yield the new proof goal R_1.

(read) Perform read/update simplification for the read-term $l\,@_x\,S$, yielding an equality $l\,@_x\,S = v'$; assume this equation for later re-use; substitute v by v' in R_1 and yield this as the new proof goal.

(cond) Introduce two new proof goals R_1 and R_2 in which e and $\neg e$ are assumed, respectively.

(wk) Introduce three proof goals R_1, $\mu \subseteq \Lambda$, and R_3, where in the latter read/update simplification is performed for R_2 and the result is assumed and a fresh state variable T is introduced about which $S \sqsubseteq_\mu T$ is assumed.

In the cases (deref), (array), (var), `pr_step` introduces two proof goals, one non-terminal R_1, and one terminal (e. g., $\bullet(l_t|\ S)$). Annotated specification expressions of (spec) are also regarded as terminal nodes. We can see that the iterated application of `pr_step` conceptually corresponds to a symbolic execution of the function.

6.5.2 Read/Update Simplification

The traversal realised by `pr_step` has one computational aspect, which is the simplification of terms depending on intermediate states. There are two kinds of such terms: let-bindings in which the defining term reads an lvalue in a program state (we also call these *read-terms*) and representation functions, which occur nested within specification expressions and not as nodes of their own in the iVC tree.

The task of *read/update simplification* is to rewrite read-terms and representation function terms in a proof state according to their update theory and hence by considering the program state in which they are evaluated and that state's relation to previous program states. A typical proof state in which read/update simplification of a read-term would be called for looks as follows:

$$\begin{aligned}&[\![\ \ldots;\ S_1(l_1 ::= e_1) = S_2;\ S_2(l_2 ::= e_2) = S_3;\\ &\ \ S_3 \sqsubseteq_{\mu_3} T_1;\ T_1(l_3 ::= e_3) = S_4;\ \ldots\ ]\!]\\ &\Longrightarrow \mathbf{let}\ v = l_4\,@_i\,S_4\ \mathbf{in}\ R_1\ v\end{aligned} \tag{6.7}$$

Because of the way that `pr_step` treats state modifications ((ext), (upd), (wk)), there is always a *current* state (here: S_4) representing the point in the function that has been reached through symbolic execution. This state is always related to the initial state (S_1) via a chain of equalities or $\sqsubseteq_\mu$ relations. In the chain all successive intermediate states are related via an extension, an update, a deallocation or w. r. t. a modification set. We have already given an example of a concrete read/update simplification for representation functions over arrays in Sec. 4.5.2. Regarding Eq. (6.7) read/update simplification means to rewrite $l_4\,@_i\,S_4$ according the update theory for integer locations (cf. Fig. 4.4 and Sec. 4.4). Assuming that $l_4 \neq l_3$ and $l_4 \notin \mu_3$, but $l_4 = l_2$, we obtain the rewrite sequence

$$\begin{aligned}l_4\,@_i\,S_4 &= l_4\,@_i\,T_1(l_3 ::= e_3) && (\textit{substitution})\\ &= l_4\,@_i\,T_1 && (\textit{read-int-update-other})\\ &= l_4\,@_i\,S_3 && (\textit{mod-read-int-eq})\\ &= l_4\,@_i\,S_2(l_2 ::= e_2) && (\textit{substitution})\\ &= e_2 && (\textit{read-int-update})\end{aligned}$$

We can thus instantiate the usual higher-order substitution rule

$$\llbracket\ s = t;\ P\ s\ \rrbracket \Longrightarrow P\ t \qquad \text{(subst)} \tag{6.8}$$

using this equality to simplify the conclusion of Eq. (6.7) to $R_1\ e_2$. We note that one occurrence of a program state is eliminated by the simplification. We also note that occurrences of read-terms are syntactically restricted to let-bindings. It it is thus clear how to isolate them from the proof state's conclusion for simplification (by β-expansion) and how to pass them to the simplification procedure described below to derive the equality. This simple substitution mechanism is also possible for representation function terms which appear nested inside specification expressions outside the scope of quantifiers.

Congruence Rules

When representation functions appear within the scope of quantifiers, a simple β-expansion cannot isolate them. Moreover, when they appear in the conclusion of an implication it might be necessary to take the premiss of the implication into account to be able to successfully perform read/update simplification on them. Consider the following example:

$$\begin{aligned}&\llbracket\ \ldots;\ S_1(l_1 ::= e_1) = S_2;\ S_2(l_2 ::= e_2) = S_3;\\ &\ \ S_3 \sqsubseteq_{\mu_3} T_1;\ T_1(l_3 ::= e_3) = S_4;\ S_4 \sqsubseteq_{\mu_4} T_2;\ \ldots\ \rrbracket\\ &\Longrightarrow \forall(i :: \mathit{nat}).\ 0 \leq i \wedge i < \mathit{len} \longrightarrow |\,\mathit{VecR}\ T_2\ (l_{3t}.[i])| < 1\end{aligned} \tag{6.9}$$

The specification expression in the conclusion is a typical one expressing that all elements of an array, interpreted as vectors, have lengths less than 1. There are two possibilities to simplify the representation function: the user can massage the proof state and isolate the term manually (here by applying universal and implication introduction rules) before calling the simplification algorithm. This can be necessary if representation functions appear deeply nested in logically complex specification expressions. For simpler cases such as the one above, however, it is more reasonable to let the system apply congruence rules during read/update simplification to accumulate further assumptions that can be used during the simplification. The method we use is akin to a simple form of window inference [133, 101]. Concretely, we only use the following two congruence rules which allow us to rewrite within universal quantifiers under the proviso that we assume an arbitrary fresh variable x as the quantified item and to rewrite within the conclusion of an implication while assuming its premiss.

$$\begin{aligned}&\llbracket\ \bigwedge x.\ P\ x = Q\ x\rrbracket \Longrightarrow (\forall x.\ P\ x) = (\forall x.\ Q\ x) &&\text{(iff-allI)}\\ &\llbracket\ P \Longrightarrow Q = Q'\ \rrbracket \Longrightarrow (P \longrightarrow Q) = (P \longrightarrow Q') &&\text{(imp-rcong)}\end{aligned}$$

Regarding Eq. (6.9) these congruences allow the algorithm to consider the term $\mathit{VecR}\ T_2\ (l_{3t}.[i])$ for an arbitrary i about which $0 \leq i \wedge i < \mathit{len}$ is assumed.

Algorithm

The algorithm for read/update simplification expects a proof goal of the form $t = ?t'$, where $?t'$ is a meta-variable that will be instantiated at the end of the algorithm and t is either a simple read-term or the full application of a representation function to its arguments. It performs three tasks: (1) it repeatedly applies theorems from the appropriate update theory for t to either reflect updates in the value that t represents (e. g., theorem *read-int-update*) or to ignore updates that do not affect t's value (e. g., theorem *read-int-update-other*), (2) in the latter case it proves the corresponding inequality conditions, and (3) it decides when to stop attempting further simplifications.

For uniformity the algorithm regards both read-terms and representation functions as functions over a location and a state, $f\ l\ S$. It also uniformly works over dependence sets (cf. Sec. 4.5), i. e. locations which affect the value of f. We denote these by $D\ l$. For read-terms ($f\ l\ S = l\,@_x\,S$) we have $D\ l = \{l\}$. The example representation function *Vec2DR* $l\ S$ has $D\ l = \{l_{\mathit{Vec2D\text{-}t}}{\to}"x", l_{\mathit{Vec2D\text{-}t}}{\to}"y"\}$. The algorithm can only deal with finite dependence sets; their shape is inferred from the update theory for the given representation function. If no arrays are involved, $D\,l$ can be brought into a normal form $\{l_1, \ldots, l_n\}$. The algorithm can also deal with arrays of structures or scalars as for *Vec2DRec-n* of Sec. 4.5.2, in which case $D\ l$ is the union of the dependence sets of the members:

$$D\ l = \bigcup_{a \leq i < b} \{l^i_1, \ldots, l^i_n\} \qquad \text{where} \quad l^i_j = l_t.[i]_t{\to}f_j \text{ or } l^i_j = l_t.[i]$$

Given $f\ l\ S$, there is either a state S' such that one of $S'(l' ::= v) = S$, $S'{\oplus}(l', t)$, $S'{\ominus}l$, or $S' \sqsubseteq_\mu S$ is assumed, or S is the initial state S_1. In the latter case, the algorithm stops and instantiates $?t'$ with $f\ l\ S$. In the former case the relation between $D\ l$ and l' or μ must be analysed. The assumption $S'{\ominus}l$ enforces a proof that l and l' have different base locations: $D\ l \cap \Uparrow_{\mathit{base\text{-}loc}\ l'} = \{\}$. (Trying to prove $l = l'$ is not useful as it corresponds to reading a deallocated location). If the proof succeeds (see the subsequent paragraph) the algorithm continues with $f\ l\ S'$; otherwise it stops and instantiates $?t'$ with $f\ l\ S$. The assumption $S'{\oplus}(l', t) = S$ also suggests to prove that $D\ l \cap \Uparrow_{\mathit{base\text{-}loc}\ l'} = \{\}$ and to continue with $f\ l\ S'$ in the case of success. The assumption $S'(l' ::= v) = S$ leaves two choices: either $l' \in D\ l$ or $l' \notin D\ l$. The algorithm tries to show the former case first. The proof is trivial if l' syntactically appears in $D\ l$; otherwise, the algorithm looks for assumptions of the form $l' = l''$ or $l'' = l'$ and tries to prove $l'' \in D\ l$.[3] For each location in $D\ l$, the algorithm knows how to reflect the update on l' in the value represented by $f\ l\ S$ by virtue of the update theory. For example, if $f\ l\ S = l\,@_i\,S$ and $l = l'$, the result is simply v, and the algorithm stops, instantiating $?t'$ with v. If $l' \in D\ l$ cannot be shown in this way, the algorithm tries to prove $l' \notin D\ l$ (see below). The remaining possible assumption is $S' \sqsubseteq_\mu S$. Since this assumption does not contain information about the values assigned to locations in μ, the algorithm must stop and instantiate $?t'$ with $f\ l\ S$ if it cannot prove $D\ l \cap \mu = \{\}$. If it can, it continues with $f\ l\ S'$.

Proving inequalities The algorithm outlined above relies on a procedure that proves statements of the two forms $l' \notin D\ l$ and $D\ l \cap X = \{\}$, where X is either a modification set μ or a term $\Uparrow_{\mathit{base\text{-}loc}\ l'}$, denoting all offsets of a base location. If $D\ l = \{l_1, \ldots, l_n\}$ the latter is equivalent to $l_1 \notin X \wedge \cdots \wedge l_n \notin X$,

[3] Such equalities can be introduced by the conditions of if- and while-statements, as in **if** (p = q) ... .

and if $D\ l = \bigcup_{a \le i < b}\{l^i_1, \ldots, l^i_n\}$ it is equivalent to $l^i_1 \notin X \wedge \cdots \wedge l^i_n \notin X$ under the assumption $a \le i < b$ for a fresh variable i. Therefore, the two cases uniformly require proofs of statements of the form $l \notin Y$. Y can be considered as being built from unions of singleton location sets such as $\{l_t{\to}f\}$, set comprehensions $\{l.\ \exists i.\ a \le i \wedge i < b \wedge l'_t.[i] = l\}$ introduced by array slices in @modifies clauses such as v[a:b], and mappings of access operators on these sets, $(\lambda l.\ l_t{\to}''f'')$ ‘ $\{l.\ \exists i.\ a \le i \wedge i < b \wedge l'_t.[i] = l\}$, introduced by chains of accesses in modifies clauses such as v[a:b].f.[4] To prove $l \notin \bigcup_{1 \le i \le m} Y_i$ means to individually prove $l \notin Y_i$. To prove $l \notin \{l.\ \exists i.\ a \le i \wedge i < b \wedge l'_t.[i] = l\}$ one can assume $a \le i \wedge i < b$ for a fresh variable i and prove $l \neq l'_t.[i]$. Mapping access operators on location sets works similarly but requires one to apply the mapping on the representative location, yielding the proof obligation $l \neq l'_t.[i]_t{\to}''f''$ for the above mapping. We can see that eventually all proof obligations can be reduced to atomic inequalities between location terms that possibly incorporate array indices about which a range assumption is made.

After reduction to atomic location inequalities, the procedure inspects each inequality in sequence. Based on the structure of the two involved location terms it decides which theorem of the memory model to apply. The possible cases are as follows:

1. Both locations are built from accesses to locations denoting distinct local or global variables, e. g., $(\nu\cdot''x''_n)_t{\to}''f'' \neq (\nu\cdot''y''_m)_{t'}.[i]$. In these cases the inequality is immediate.

2. The top-level access on both terms is a field selection and distinct fields are accessed, e. g. $l_t{\to}''f'' \neq l'_{t'}{\to}''g''$. The memory model allows to infer that the locations are unequal if both terms denote valid locations in *some* state.

3. The top-level access on both terms is a field selection on the same field. Since field selection is injective, the procedure continues with proving the accessed locations themselves unequal.

4. The top-level accesses are a field selection and an array access, respectively; the memory model ensures their distinctness if both denote valid locations.

5. The top-level accesses are both array accesses, e. g. $l_t.[i] \neq l'_{t'}.[j]$. The procedure can only prove the locations unequal if both are valid ones. In the case that $t \neq t'$ the inequality is immediate. Otherwise, the procedure delegates to Isabelle/HOL's linear integer arithmetic decision procedure to derive whether $i = j$ under the assumptions of the current proof state. If it derives $i = j$, the locations are unequal if and only if $l \neq l'$. If it derives $i \neq j$, the locations are certainly unequal. If it fails, the inequality cannot be decided.

6. One of the locations is the value of a valid pointer whose value cannot be associated with a concrete location (an *indeterminate location*). This is the case if it is the result of reading a global pointer variable in the initial state ($\gamma{\cdot}g \mathbin{@_l} S_1$) or if it is a pointer argument value ($v :: \mathit{Loc}$). If the other

[4] f ‘ X is the image of f under X, i. e., $\{y.\ \exists x \in X.\ f\ x = y\}$

location is based on a local variable then the inequality is immediate, since a valid location cannot point to a fresh location. For example, it is true that $g \mathbin{@_l} S_1 \neq \nu{\cdot}l_n$ given the assumptions $\nu{\cdot}l_n \notin_S S_1$ and $\bullet((g \mathbin{@_l} S_1)_t|\ S_1)$.

If the other location is based on an indeterminate location as well, the inequality must be derived from existing assumptions. The procedure can deal with two cases. We assume the inequality to be derived is $v \neq v'$.

(a) The precondition of the function to be verified, or the postconditions of functions called during symbolic execution, or the conditions of the taken branches asserted facts which from which $v \neq v'$ is implied as a propositional tautology.

(b) The precondition contains a separation constraint from which a binary assertion $S_1 \models L\ X \between L\ X'$ can be derived by the rules described in Sec. 3.4.2 such that $v \in X$ and $v' \in X'$.[5] By the definition of satisfaction of separation constraints this implies that $v \neq v'$.

    An example for this case is a function specification containing the separation constraint `@memory *a <*> *b`. If `a` and `b` are global integer pointers, the constraint gets translated to

$$S_1 \models L\ \{\gamma\cdot\text{"}a\text{"} \mathbin{@_l} S_1\} \between L\ \{\gamma\cdot\text{"}b\text{"} \mathbin{@_l} S_1\}$$

    and allows the procedure to derive that the two mentioned location terms denote unequal locations.

6.5.3 Further Tactics

Besides `pr_step` the verification environment provides other noteworthy tactics which we briefly describe here.

1. `prtac` is the tactic by which the iVC can be generated from the initial proof obligation *correct-function-modular* Θ *fid*. It also performs the simplifications described in Sec. 6.4.2 and conceals a substantial amount of technicalities from the verifier, such as looking up identifiers in environments, expanding specifications of called functions, evaluating the semantics of modification sets, and several more.

2. `pr_valid` is used to automatically prove validity proof goals of the form $\bullet(l_t|\ S)$ or $\bullet[l_t|\ n|\ S]$ using the rules of Sec. 4.2.6. It is immediately clear that this tactic cannot be a decision procedure, i. e., cannot be complete, since it is in general undecidable whether a pointer or array access is always valid. In practice, however, intervention by the verifier was only necessary when the index expression of an array access involved non-linear arithmetic, as in $\bullet[a_t|\ (i+j) \operatorname{div} 2|\ S]$. In these cases he needs to aid the tactic by deriving the fact that $0 \leq (i+j) \operatorname{div} 2 \leq n$, where $\bullet[l_t|\ n|\ S']$ is known for some state S' related to S via updates or modification sets. In contrast, when knowing that `i` is a valid index for array `a`, the tactic successfully derives the validity of a pointer required by a called function's precondition, as in `foo(&a[i])`, both if `a` is a function local array as well as if its own validity is assured by the calling function's precondition.

[5] Here, L is the constructor for the datatype *mem-descr* and $L\ X$ and $L\ X'$ are memory descriptions of type *Loc mem-descr*. We recall that $S \models M \between M'$ only holds if the domain of S is the disjoint union of the locations described by M and M'.

3. `pr_simp` is the tactic for explicitly requesting read/update simplification in an arbitrary proof state. In the standard case, the tactic looks for occurrences of representation functions and read-terms both in the assumptions of the current proof goal as well as in the conclusion. For each such term t it extracts a subgoal of the form $t = ?t'$ and applies the algorithm described above to this equation. t is then substituted by the obtained instantiation for $?t'$ in the proof state. The tactic can be configured to only select terms matching certain criteria, such as specific variable names or program states.

Chapter 7

Verification of the SAMS Code

In the preceding chapters the verification environment for the functional verification of C programs in Isabelle/HOL was presented. In this chapter it is demonstrated how the environment was used in a concrete certification effort. This is done from two different perspectives; first examples of concrete functions that were implemented in the SAMS project as well as their specifications are discussed and interesting aspects of their verification are pointed out. Afterwards —beginning with Sec. 7.5— the integration of this environment into the broader verification and validation process that is required by IEC 61508-3 is worked out.

7.1 Algorithm for Computing Safety Zones

We have formally verified the algorithm responsible for computing safety zones, which is part of the overall safety-related software developed in the SAMS project. Before we can discuss its verification, a brief description of its workings is in order.

A laser scanner is a sensor that measures the distance to obstacles in its surroundings by emitting laser beams in a fixed angular range and measuring the time until the reflection of each emitted beam returns (*time of flight measurement*). Since the running time is proportional to the distance of the reflecting object, the latter can be computed from the former. A laser scanner was used in the safety component developed in the SAMS project. The safety component's function is to raise a signal whenever objects are detected as being "too close" to the equipment under control (EUC). In our case the EUC reacting to the component's signals is a moving vehicle such as an automated guided vehicle (AGV). Here, the component's function is to raise an emergency stop signal to ensure collision avoidance. The definition of "too close" depends on the application scenario and is configured manually for each such scenario in most state-of-the-art scanner-based safety components.[1]

[1] For example, the safety laser scanners *ROTOSCAN RS4* by Leuze electronic (`http://www.leuze.de`) or the *S3000* by SICK (`http://www.sick.com`)

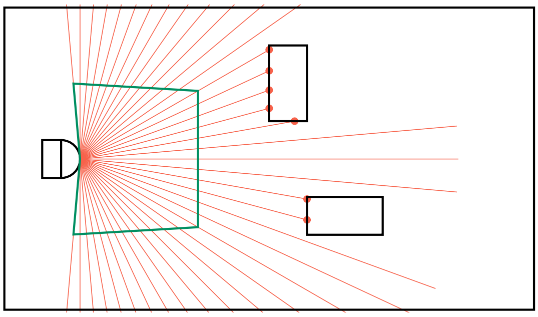


Figure 7.1: Safety zones in state-of-the-art safety laser scanners. The green polygon represents a safety zone.

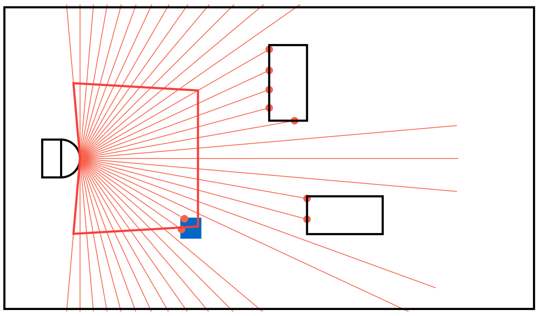


Figure 7.2: The safety zone is violated by the blue object.

Definition 7.1. A *safety zone* is the area monitored by the safety component in which detected objects lead to a safety-related reaction of the component, namely the emission of an emergency stop signal. A safety zone is determined by the minimum distances that must be measured by the scanner in every cycle.

Fig. 7.1 depicts a scanner emitting beams, some of which are reflected by obstacles[2], and visualises an (arbitrarily chosen) safety zone. Fig. 7.2 depicts the same situation, but with an additional object that it detected inside the safety zone (a safety zone *violation*).

7.1.1 Braking Model

The novel aspect of the software developed in the SAMS project is that the monitored safety zones for the EUC are chosen depending on the *current velocity* of the EUC. To enable this, the safety zones are computed by a microcontroller of the safety component during operation. This is an obvious improvement over a fixed, manually configured set of safety zones, since for safety reasons these have to be conservatively defined and will hence be too large for most velocities, impairing the flexible movement of the AGV. The computed safety zone for a given (imprecise measurement of the) current velocity must be a superset of the *braking area* which is the area covered by the EUC during braking. In

[2] Some beams are not reflected at all, which represents the case where the maximum range of the scanner is exceeded before the beam reaches an object.

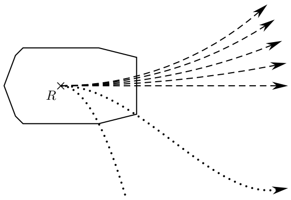


Figure 7.3: The braking model allows circular and straight trajectories (indicated by dashed lines; dotted lines show examples of invalid braking behaviour)

reality, the braking area is not merely a function of the velocity of the EUC, but also depends on physical properties such as the friction between the wheels of the EUC and the surface it drives on, the strength of its mechanical brakes, and many others. We developed a simplified *braking model* that enables us to reason about the braking behaviour of vehicles that are typically used in industrial settings. The braking model is defined in a two-dimensional model of the world; this is reasonable since laser scanners only sense their environment in two dimensions and because industrial AGVs usually drive on flat ground. The fundamental assumption of the braking model is that the braking trajectory of the EUC is either a straight line or a circular arc. An intuitive example is a car that brakes with a fixed steering and without slipping or sliding. This is depicted in Fig. 7.3.

The braking model must be capable of determining a safe approximation of the braking distance that a concrete EUC will cover for a given velocity $v \leq v_{max}$, where v_{max} is the maximum velocity of the EUC. More precisely, a *braking configuration* (s, α) must be determined:

Definition 7.2. A *braking configuration* (s, α) for a given forward velocity v and angular velocity ω defines the length s and the angle α of the arc that describes the braking trajectory covered by the EUC while performing an emergency stop at velocity (v, ω) according to the braking model. As a special case, $(s, 0)$ defines a straight line of length s.

To this end, the braking model is parameterised over a number of braking measurements, $(v_1, s_1), \ldots, (v_{max}, s_{max})$, which must be determined before putting the EUC into operation. Here, s_i is the braking distance covered by the EUC during an emergency stop at velocity v_i in straight forward movement. These are used to generate a piecewise linear function which yields an over-approximation of the actual braking distance for any forward velocity $v_1 \leq v \leq v_{max}$.[3] Fig. 7.4 depicts two possible linear approximations of a convex braking function interpolated from one or two braking measurements, respectively. The braking distance covered when cornering is extrapolated from the braking distance at forward movement. The energetic considerations on which the extrapolation is based are described in Sec. 7.2.1 further below. The strategy is to first compute a velocity v_G for straight forward motion that in-

[3] Technically, the function will yield an over-approximation only if the actual braking behaviour of the EUC is described by a convex function from velocities to distances, which is the case for all relevant vehicles.

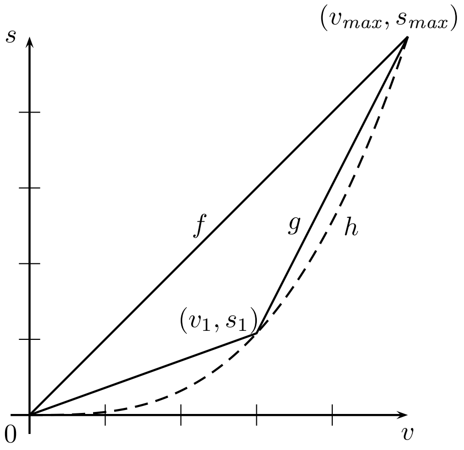


Figure 7.4: Two piecewise linear functions f and g approximate the actual braking function h (shown as a dashed line). g makes use of an additional sampling point (v_1, s_1)

duces at least as much kinetic energy as is present at the given velocity (v, ω), and to use v_G for the computation of s. The value of α then follows immediately.

Another assumption is that all objects in the surroundings of the EUC are *stationary*, which is a common assumption for collision avoidance in industrial settings. While persons are obviously not stationary, one can expect that they at least do not actively approach the EUC during its operation. Finally, it is assumed that the braking behaviour of the EUC is independent of time and its current location.

7.1.2 Computation of Safety Zones

When the EUC and the safety component are in operation, the scanner cyclically yields a sequence of distance measurements in its angular range. For example, the scanner that was used in the SAMS project[4] yields up to 529 distance measurements in a range of 190° every 40ms. Within these 40ms the safety software has to determine whether it is safe for the EUC to continue travelling, or whether an emergency stop signal must be raised. This procedure is repeated for all sequences of measurements. The scanner's cycle time thus defines the clocking of the safety software. Each sequence is measured while the EUC travels at its *current velocity* (v, ω). These measurements are immediately used by the safety software to detect violations of the currently required safety zone. The latter, in turn, is determined by (v, ω) according to the braking model. The task of the software module considered in this chapter is to compute these safety zones.

In each cycle, a safety zone is computed that is safe for the current velocity of the EUC. The algorithm expects the following inputs: (1) a velocity interval

[4] ROTOSCAN RS4 by Leuze electronic

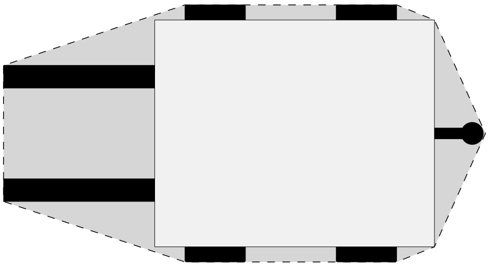


Figure 7.5: The shape of the EUC is approximated by its convex hull

that safely approximates the actual current velocity: $(v, \omega) \in [v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$; (2) a finite set of points $[R_i]_{i=1}^{n}$ representing the convex hull of the contour of the EUC (cf. Fig. 7.5); (3) the abovementioned braking measurements $(v_1, s_1), \dots, (v_{max}, s_{max})$ from which the braking behaviour can be determined by the braking model; and finally (4) a latency T_l that safely approximates the duration until a potential emergency stop takes full effect, i. e., a time in which the vehicle continues moving at its current velocity. It proceeds as follows:[5]

First, the input velocity interval $[v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$ is transformed into the braking configuration area $[s_{min}, s_{max}] \times [\alpha_{min}, \alpha_{max}]$. The transformation *braking-configuration*(v, ω) describes a movement consisting of moving for time T_l with velocity (v, ω), and then braking on a circular arc that retains the radius defined by v and ω.

[Step 1] For (v, ω) in $\{v_{min}, v_{max}\} \times \{\omega_{min}, \omega_{max}\}$, compute the braking configuration (s, α) as follows, and determine minimum and maximum s_{min}, s_{max}, α_{min}, and α_{max} of the four results:

$$(s, \alpha) = \textit{braking-configuration}(v, \omega) \tag{7.1}$$

Then, compute the safety zone in terms of a finite set of points $[P_k]_{k=1}^{K}$ and a buffer radius q. The safety zone is an area $A^+\left([P_k]_{k=1}^{K}; q\right)$, given by the union of the convex hull of $[P_k]_{k=1}^{K}$ and the set of all points having distance of at most $q > 0$ to any point of that convex hull, a construction called the Minkowski sum:

$$A^+\left([P_k]_{k=1}^{K}; q\right) = \left\{P + Q \;\middle|\; P \in \operatorname{conv}\left\{[P_k]_{k=1}^{K}\right\}, |Q| \leq q\right\} \tag{7.2}$$

[Step 2a] To compute points $[P_k]_{k=1}^{K}$, approximate the braking trajectories for all points of the convex hull $[R_i]_{i=1}^{n}$: For all $(s, \alpha) \in \{s_{min}, s_{max}\} \times \{\alpha_{min}, \alpha_{max}\}$, compute

$$H_{s,\alpha} = \left\{[U_{i,s,\alpha}^{1}, U_{i,s,\alpha}^{3}, V_{i,s,\alpha}^{0}, \dots, V_{i,s,\alpha}^{L-1}]_{i=1}^{n}\right\},$$

where $U_{i,s,\alpha}$ and $V_{i,s,\alpha}$ are given as follows (for i in $1, \dots, n$):

$$\begin{aligned} U_{i,s,\alpha}^{1} &= R_i & U_{i,s,\alpha}^{2} &= T(\tfrac{s}{L}, \tfrac{\alpha}{L}) \cdot R_i \\ U_{i,s,\alpha}^{3} &= T(s, \alpha) \cdot R_i \\ V_{i,s,\alpha}^{0} &= U^1 + Q(\tfrac{\alpha}{L})\tfrac{1}{2}(U^2 - U^1) & V_{i,s,\alpha}^{j} &= T(\tfrac{j \cdot s}{L}, \tfrac{j \cdot \alpha}{L}) \cdot V^0 \end{aligned} \tag{7.3}$$

[5] The subsequent text of subsection 7.1.2 is a slightly adapted and extended version of Sec. 2.1 of [150] which is in major parts due to the author's colleague Holger Täubig.

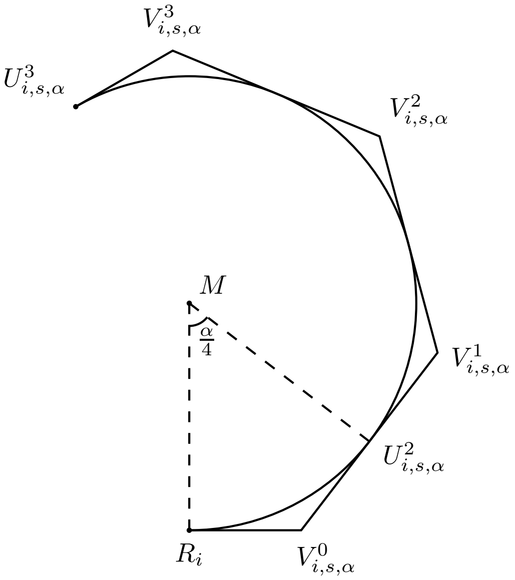


Figure 7.6: Approximation of an arc by start- and endpoint and $L = 4$ auxiliary points

with

$$T(s, \alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & s \operatorname{sinc}\frac{\alpha}{2}\cos\frac{\alpha}{2} \\ \sin\alpha & \cos\alpha & s \operatorname{sinc}\frac{\alpha}{2}\sin\frac{\alpha}{2} \\ 0 & 0 & 1 \end{pmatrix}$$

and

$$Q(\alpha) = \begin{pmatrix} 1 & \tan\frac{\alpha}{2} & 0 \\ -\tan\frac{\alpha}{2} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

This computation approximates the arc determined by (s, α) by its endpoints (R_i and $U^3_{i,s,\alpha}$) and L auxiliary points V^j. It can be easily understood graphically; we give an example where $L = 4$ in Fig. 7.6. The auxiliary points are simply the intersections of tangents of the arc that are set $\frac{\alpha}{L}$ angular units apart.

Now, $[P_k]_{k=1}^K$ is the result of a standard convex hull algorithm like Graham scan applied to the union of the $H_{s,\alpha}$ for all $(s, \alpha) \in \{s_{min}, s_{max}\} \times \{\alpha_{min}, \alpha_{max}\}$.

[Step 2b] The buffer radius q includes a conservative error approximation for the algorithm. In particular, it incorporates the error introduced by transforming the velocity interval $[v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$ into $[s_{min}, s_{max}] \times [\alpha_{min}, \alpha_{max}]$. This error is associated with the fact that the true s_{max} and α_{max} are not determined by applying *braking-configuration* to v_{max} and ω_{max}, but by some $(v, \omega) \in [v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$. We elide the details here, but point out that the correctness of the definition of q given in Eq. (7.4) has been formally verified, meaning that the safety zone $A^+\left([P_k]_{k=1}^K; q\right)$ contains all braking trajectories of all points of the convex hull $[R_i]_{i=1}^n$ for all velocities $(v, \omega) \in$

$[v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$. q is given by the equation

$$q = \frac{1}{6}\left(\frac{\alpha_{\max}-\alpha_{\min}}{2}\right)^2 \max\{|s_{\max}|;|s_{\min}|\} + \left(1-\cos\frac{\alpha_{\max}-\alpha_{\min}}{2}\right)\max_{1\leq i\leq n}\{|R_i|\}. \tag{7.4}$$

[Step 3] Finally, the computed points $[P_k]_{k=1}^K$ are transformed into scanner coordinates, and the safety zone $A^+\left([P_k]_{k=1}^K; q\right)$ is sampled into a laser-scan like representation, defining a minimum length for each laser beam which has to be measured in order to ensure that no obstacle is inside the safety zone. We elide the details of the sampling subroutine, because it is not part of the subsequent discussion about the domain modelling and the verification of the algorithm.

The guarantee that the algorithm gives about the computed safety zone for a velocity interval $[v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$ is that the EUC stops within the safety zone if it travels for latency time T_l with constant velocity $(v, \omega) \in [v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$ and then brakes according to the braking model. Its correctness relies on two properties whose verification we will examine in the following: the correctness of the computation of (s, α) according to the braking model and a conservative approximation of the area $A^+\left([P_k]_{k=1}^K; q\right)$.

7.2 Domain Modelling

We show two fragments of the domain model which are used in the specification and verification of two exemplary functions in Sec. 7.3 that form a crucial part of the overall algorithm.

The overall domain modelling consists of 11 Isabelle theory files, about 110 definitions and 510 theorems. The domain was largely developed in about five months by a mathematician without prior knowledge of Isabelle, with contributions by the author of this thesis. We consider this a point supporting the argument that the key competency when working with Isabelle is a strong background in mathematics and particularly in formal logic. Moreover, the theorem prover Isabelle is mature enough a tool so that the necessary technicalities accompanying its use do not overly distract from or impede a formal development.[6]

7.2.1 Formalisation of the Braking Model

Preliminary Definitions

We employ Isabelle's expressive type definition facilities to introduce a type *SO2* of *angles* that is isomorphic to the set of radian angles $\{\phi :: \mathit{real}.\ 0 \leq \phi < 2\pi\}$. Using this type instead of the larger type *real* allows us to omit side-conditions in theorems involving angles.

For technical reasons, it is most convenient to use quotient sets as the carriers of newly defined types. We can use the set of all real numbers quotiented by the natural 'modulo 2π' equivalence relation between angles, since it is equivalent to the above set of all angles:

[6] We decided not to translate the domain formalisation described subsequently and use the original German terms instead. This way we can show the definitions and theorems in the form in which they were presented to the certification authority TÜV Süd Rail GmbH. An exception is the use of `@memory` clauses, which were not yet implemented at the time of certification, where equivalent specifications using `\unrelated` and `\separated` were used.

```
definition
SO2_rel :: "(real * real) set"
where
"SO2_rel = {(a, b) . ∃ k::int . a = b + 2 * pi * real k}"
```

```
typedef (SO2)
SO2 = "UNIV//SO2_rel"
```

We recall that the quotient $X//R$ is defined as $\bigcup_{x \in X} \{\{y.\ R\,x\,y\}\}$. For an equivalence relation, such as *SO2-rel*, this is the set of all its equivalence classes. Along with the above definition, Isabelle defines a function *Rep-SO2* :: *SO2* ⇒ *real set* yielding all real numbers in the equivalence class of the given angle. It has a partial inverse function *Abs-SO2* :: *real set* ⇒ *SO2* that abstracts equivalence classes of real numbers. We define a constructor function from angles given as real numbers:

```
definition
SO2_C :: "real ⇒ SO2"
where
"SO2_C a = Abs_SO2 (SO2_rel``{a})"
```

The movement of vehicles can be described by *rigid body transformations*, which are affine transformations with the additional property that the Euclidean distance between any two transformed points equals the distance of the original points. Such transformations are given by a translational and a rotational part:

```
types
  SKT = "Punkt * SO2"
```

where *Punkt* is identical to the type *complex* of complex numbers and represents two-dimensional points. Again, a constructor function comes in handy:

```
definition
SKT_C :: "Punkt ⇒ real ⇒ SKT"
where
"SKT_C p α = (p, SO2_C α)"
```

By using complex arithmetic we can concisely define the application of a rigid body transformation to a given point. It is well-known that rotation coincides with complex multiplication by unit vectors and that translation coincides with complex addition. Mathematically, if we are given a rotation angle ϕ and a translation vector (z, w), the transformation of rotating a point (x, y) by ϕ and translating the result by (z, w) is given by $(\cos(\phi), \sin(\phi)) \cdot (x, y) + (z, w)$. Since Isabelle allows us to overload operators, we can adopt this literally in the definition of the transformation function:

```
definition
transformiere :: "SKT ⇒ Punkt ⇒ Punkt"
where
"transformiere H p = (holeRotation H) * p + (holeTranslation H)"
```

While it is trivial to extract the translational part of the rigid body transformation H (*holeTranslation* simply yields the first component of the tuple), the rotational part of H of type *SO2* must be converted to a complex number with an absolute value (*modulus*) equal to 1 and an *argument* equal to the angle ϕ represented by the *SO2* value. This is done via *holeRotation.*

```
definition
```

```
rotation :: "real ⇒ complex"
where
"rotation a = Complex (cos a) (sin a)"
```

definition

```
holeRotation :: "SKT ⇒ complex"
where
"holeRotation H = contents (rotation ‘ Rep_SO2 (snd H) )"
```

lemma *rotation_Rep_SO2_singleton:*

```
" ∃x. rotation ‘ Rep_SO2 a = {x}"
```

We turn the set of equivalent angles into a singleton by mapping *rotation* on it (using the facts that $\sin(\phi) = \sin(\phi + 2\pi)$ and $\cos(\phi) = \cos(\phi + 2\pi)$ in the proof of lemma *rotation-Rep-SO2-singleton*), and extract the singleton's sole element via *contents*, a pre-defined function satisfying *contents* $\{x\} = x$.

Arc Transformations

The fundamental rigid body transformation we are interested in is the transformation along an arc or along a straight line, since these are used to describe the braking behaviour of vehicles in the model. The coordinate system used to describe a vehicle's movement has its origin at the centre of the axle that points towards the centre of the circle described by the driven curve. We call this point the vehicle's *reference point*. In the standard case of a four-wheeled vehicle with steered front axle this is the centre of the back axle, as depicted in Fig. 7.7.

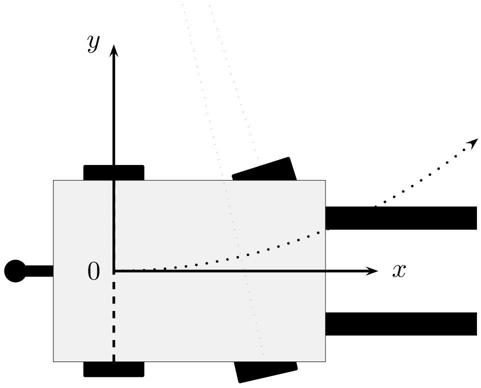


Figure 7.7: Coordinate system for a four-wheeled vehicle with steered front axle

The first task is to compute the coordinates (x, y) of the reference point given that it travels a braking distance of s units along an arc whose angle is α. (How the braking model yields (s, α) for a given pair of forward and angular velocity (v, ω) is described further below.) Together with the angle α these coordinates induce the rigid body transformation that transforms any point of the vehicle contour to its corresponding end position. The situation is depicted in Fig. 7.8. The final orientation of the vehicle is equal to α, the angle of the travelled arc. (x, y) is the end position of the vehicle's reference point. Euclidean geometry tells us that the length of the chord a equals $2R\sin\frac{\alpha}{2}$ and that $s = \alpha R$. Thus,

$$a = s\frac{\sin\frac{\alpha}{2}}{\frac{\alpha}{2}} = s\,\mathrm{sinc}(\frac{\alpha}{2}), \quad \text{where } \mathrm{sinc}(x) = \begin{cases} \frac{\sin(x)}{x} & \text{if } x \neq 0 \\ 1 & \text{if } x = 0 \end{cases} .$$

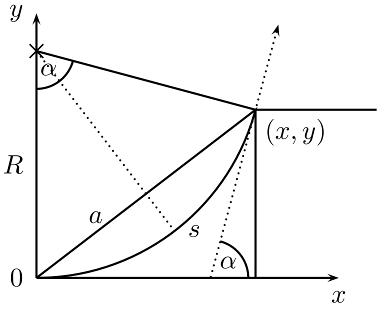


Figure 7.8: Geometric construction of the stop position (x, y)

The sinc function (for *sinus cardinalis*) is the continuous extension of $\frac{\sin(x)}{x}$ and allows us to use the same formula for the case where the vehicle brakes during a straight forward motion, where $(x, y) = (s, 0)$. Moreover, the angle between the x-axis and the chord a equals $\frac{\alpha}{2}$. Hence, we obtain

$$\begin{pmatrix} x \\ y \end{pmatrix} = s\,\mathrm{sinc}\,\frac{\alpha}{2}\begin{pmatrix} \cos\frac{\alpha}{2} \\ \sin\frac{\alpha}{2} \end{pmatrix} \tag{7.5}$$

We are finally in a position to formally define the rigid body transformation in terms of an arc length s and an arc angle of α. We call this an *arc transformation.* It is defined by a translational part given by Eq. (7.5) and a rotational part of α.

```
definition
bogentransformation :: "real ⇒ real ⇒ SKT"
where
"bogentransformation s α =
   SKT_C ((s*sinc (α/2)) *R (rotation (α/2))) α"
```

In the following, the *parameters* s, α of arc transformations are always those determined by the motion of the vehicle's reference point. An arbitrary point p of the vehicle's contour can now be transformed to its end position after braking by applying the arc transformation. The end position of p is given by *transformiere* (*bogentransformation s* α) *p*.

Approximating the Braking Distance

Next, we need to formalise the piecewise linear approximation of the actual convex braking function, to obtain (s, α) for a given velocity measurement (v, ω). As explained in Sec. 7.1.2, we may assume the existence of braking measurements $(v_1, s_1), \ldots, (v_n, s_n)$ where each s_i denotes the braking distance covered by the vehicle when braking at velocity $(v_i, 0)$, i. e. straight forward motion.

Here we assume that the v_i are ordered and that v_1 is the maximum value and that $v_n = s_n = 0$. The approximation of the braking distance s for a velocity v with $v_i > v \geq v_{i+1}$ is determined by

$$s = \frac{s_i - s_{i+1}}{v_i - v_{i+1}}(v - v_{i+1}) + s_{i+1} \tag{7.6}$$

In the case that $v > v_1$ we define a cubic over-approximation of the braking function, which has been determined as safe for all braking behaviours in which greater velocities do not result in a decrease of braking power.

$$s = \frac{s_1}{(v_1)^3}\, v^3 \tag{7.7}$$

In contrast to the program code, where the lookup of i has to be efficient, we can *specify* the linear approximation in a simple fashion whose correctness can easily be validated. The primitive recursive function *bremsweg-approx-fkt* expects a list of (v_j, s_j) pairs in decreasing order w. r. t. v_j and performs the respective computation of Eq. (7.6) or Eq. (7.7).

```
fun
bremsweg_approx_fkt :: "(real * real) list ⇒ real ⇒ real"
where
"bremsweg_approx_fkt [(vm, sm)] v = (sm / vm^3) * v^3 " |
"bremsweg_approx_fkt ((v1, s1) # (v2, s2) # z) v =
    (if v1 ≤ v then bremsweg_approx_fkt [(v1, s1)] v
     else if v2 ≤ v then (s1 - s2) / (v1 - v2) * (v - v2) + s2
     else bremsweg_approx_fkt ((v2, s2) # z) v)"
```

bremsweg-approx-fkt can be applied directly to compute s if $\omega = 0$. For motion along a curve, where $\omega \neq 0$, s is over-approximated by applying *bremsweg-approx-fkt* to a straight forward velocity v_G at which the kinetic energy of the EUC is at least as large as when travelling at velocity (v, ω). The brief physical derivation of v_G goes thus: the kinetic energy of an object moving at velocity (v, ω) is given by $\frac{1}{2}mv^2 + \frac{1}{2}J\omega^2$, where m is the object's mass and J its moment of inertia; J is bounded by the *maximum extension* D of the object (the maximum distance of any two points occupied by the object) via $J \leq D^2 m$; this yields a bound on the object's kinetic energy as $\frac{1}{2}m(v^2 + D^2\omega^2)$; by letting $v_G = \sqrt{v^2 + D^2\omega^2}$, the desired straight forward velocity is found. This is formalised in function *aequiv-v-gerade*:

```
definition
aequiv_v_gerade :: "real ⇒ real ⇒ real ⇒ real"
where "aequiv_v_gerade v ω D = sqrt (v^2 + (D * ω)^2)"
```

Once s is computed, α is given by $\alpha = (s/v)\,\omega$, because braking on a circular arc entails that v/ω is constant throughout and equal to the circle's radius R, which in turn equals s/α.

We conclude this section with a validation that *bremsweg-approx-fkt* is defined correctly. The criterion is that it should yield a convex function if applied to a list of braking measurements that are given in decreasing order and which themselves define a sub-linear list:

```
fun
sublineare_liste :: "(real × real) list ⇒ bool"
where
```

```
"sublineare_liste [(x1, y1), (x2, y2)] = (y2 ≤ y1)" |
"sublineare_liste ((x1, y1) # (x2, y2) # (x3, y3) # z) =
  ((y2 - y1) ≤ (x2 - x1) / (x3 - x1) * (y3 - y1) ∧
  sublineare_liste ((x2, y2) # (x3, y3) # z))"
```

A sub-linear list contains pairs (x_i, y_i) in which the slope between the pairs (x_i, y_i) and (x_{i+2}, y_{i+2}) (given by $\frac{y_{i+2}-y_i}{x_{i+2}-x_i}$) is greater or equal to the slope between (x_i, y_i) and (x_{i+1}, y_{i+1}). This criterion can also be used to validate that concrete braking measurements do not contradict the assumption of a convex braking function: a convex braking function will lead to measurements that constitute a sub-linear list. The convexity of a function between boundaries x_0 and x_N can be defined in a similar vein:

```
definition konvexe_fkt :: "(real ⇒ real) ⇒ real ⇒ real ⇒ bool"
where
"konvexe_fkt f x0 xN =
   (∀x1 x2 x3. x0 ≤ x1 ∧ x1 ≤ x2 ∧ x2 ≤ x3 ∧ x3 ≤ xN ⟶
    (f x2 - f x1) * (x3 - x1)  ≤ (f x3 - f x1) * (x2 - x1))"
```

Noting that *sortiert-nach* P is a predicate over lists asserting that consecutive elements e_i, e_{i+1} in the list satisfy $P\ e_i\ e_{i+1}$, it can be shown that *bremsweg-approx-fkt* yields a convex function between 0 and v_1, the maximum velocity measurement.

```
lemma bremsweg_approx_fkt_konvex:
"⟦ sublineare_liste xs;
   sortiert_nach (λx y. fst x > fst y) xs;
   xs = (v1, s1) # b # list; last xs = (0, 0)⟧
 ⟹ konvexe_fkt (bremsweg_approx_fkt xs) 0 v1"
```

The proof of this theorem is by induction over *list*, the tail of the tuple-list *xs*. It rests on several properties of convex functions, such as the fact that the boundaries in which a function is convex can be concatenated:

```
lemma konvexe_fkt_zusammengesetzt:
"⟦a≤b; b<c; c≤d; konvexe_fkt f a c; konvexe_fkt f b d⟧
   ⟹ konvexe_fkt f a d"
```

or the fact that the mapping from list indices to the first component of the list's elements is a monotone function:

```
lemma sortiert_nach_gr:
 "(∀a b. a < b ⟶ b < length bs ⟶
     sortiert_nach (λ(v::real, s::real) (v', s'). v' < v) bs ⟶
       fst (bs ! a) ≤ v  ⟶ fst (bs ! b) < v)"
```

7.2.2 Arc Approximation

The domain formalisation is furthermore concerned with the approximation of arcs by polylines, and the covering of arcs by the convex hulls of these polylines. Arc coverings play an important role in the computation of safety zones. In this section we illustrate the relevant concepts as well as the proof that the polyline defined through auxiliary points as shown in Fig. 7.6 covers the associated arc.

Arcs

The necessity for computing arc coverings arises from the fact that the braking area covered by a vehicle is defined by the union of the arcs described by all contour points of the vehicle. A conservative approximation (i. e., a superset) of this union yields the basis for a safe and correct safety zone. The arcs described by contour points are given by the *trace* of arc transformations, in the following way. We define the end point of an arc as the application of an arc transformation:

```
definition
bogenendpunkt :: "real ⇒ real ⇒ Punkt ⇒ Punkt"
where
"bogenendpunkt s α p = transformiere (bogentransformation s α) p"
```

and then define an arc (*bogen*) from a point p to its end point *bogenendpunkt* s α p as the set of all points reached by scaling the transformation between 0 and 1:

```
definition
bogen :: "real ⇒ real ⇒ Punkt ⇒ Punkt set"
where
"bogen s α p =
   {p'. (∃a. 0 ≤ a ∧ a ≤ 1 ∧ p' = bogenendpunkt (a*s) (a*α) p)}"
```

We call a superset X of an arc, *bogen* s α $p \subseteq X$, an *arc covering.* Computed arc coverings describe convex sets, a concept we briefly recapitulate next.

Convex Sets of Vectors

A set X of two-dimensional points, or vectors, is *convex* if for each pair of vectors $v_1, v_2 \in X$, the line connecting them, which is given by $\{t \cdot v_1 + (1-t) \cdot v_2 \mid 0 \leq t \leq 1\}$, is also contained in X. Formally:

```
definition
konvex :: "Punkt set ⇒ bool"
where
"konvex K = (∀x∈K. ∀y∈K. ∀t. (0≤t ∧ t≤1) ⟶
                (t *R x + (1 - t) *R y) ∈ K)"
```

The *convex hull* of a set of vectors X is the smallest convex superset of X. Since the superset relationship defines a complete lattice, it is equal to the intersection of all convex supersets:

```
definition
konvexe_huelle :: "Punkt set ⇒ Punkt set"
where
"konvexe_huelle X = ⋂{K . konvex K ∧ X⊆K}"
```

We note that several standard properties of convex hulls can be proven automatically by Isabelle. These include the fact that a set X is a subset of its convex hull; that the convex hull operator behaves as an identity on convex sets; that it is a monotonic and idempotent operator; or that the union of the convex hulls of a set of vector sets X_i is a subset of the convex hull of the union of the X_i.

```
lemma subset_konvexe_huelle:
```

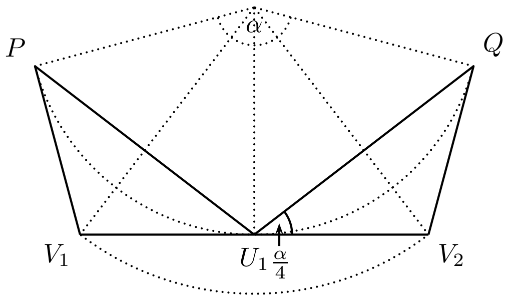


Figure 7.9: Construction of points V_i to obtain triangles covering arc segments

```
"X ⊆ konvexe_huelle X"

lemma konvex_imp_konvexe_huelle_eq:
"konvex K ⟹ konvexe_huelle K = K"

lemma konvexe_huelle_monoton:
"X ⊆ Y ⟹ konvexe_huelle X ⊆ konvexe_huelle Y"

lemma konvexe_huelle_idem:
"konvexe_huelle (konvexe_huelle X) = konvexe_huelle X"

lemma konvexe_huelle_Union:
"⋃(konvexe_huelle ‘ X) ⊆ konvexe_huelle (⋃X)"
```

Arc Coverings

Every arc with an angle less than π is covered by the triangle defined by the arc's end points and the intersection of the tangents through these end points. This situation is depicted in Fig. 7.9, where the arcs with end points P and U_1 as well as U_1 and Q, both with an angle of $\frac{\alpha}{2}$, are contained in the triangles constructed with the intersection points V_1 and V_2, respectively. The latter two can easily be constructed if the points are regarded as Euclidean vectors; e. g., V_1 is the sum of P, half the direction vector $P - U_1$, and the altitude (vector) of the isosceles triangle $\triangle P\, V_1\, U_1$, two of whose angles are known to be $\frac{\alpha}{4}$:

$$\begin{aligned}
V_1 &= P + \frac{1}{2}(P - U_1) + \tan\left(\frac{\alpha}{4}\right) \cdot \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \cdot \frac{1}{2}(P - U_1) \\
&= P + \frac{1}{2} \cdot \begin{pmatrix} 1 & \tan(\frac{\alpha}{4}) \\ -\tan(\frac{\alpha}{4}) & 1 \end{pmatrix} \cdot (P - U_1) \\
V_2 &= P + \frac{1}{2} \cdot \begin{pmatrix} 1 & \tan(\frac{\alpha}{4}) \\ -\tan(\frac{\alpha}{4}) & 1 \end{pmatrix} \cdot (U_1 - Q).
\end{aligned}$$

These equations are directly adapted (once again disguised as complex arithmetic operations) in the definition of function *konvexpunkt* which yields the intersection point, or *convex point*, given start point S, end point E, and the angle α of the arc from S to E. Using the variable names of Fig. 7.9, we therefore have *konvexpunkt* $P\ U_1\ \left(\frac{\alpha}{2}\right) = V_1$.

```
definition
konvexpunkt :: "Punkt ⇒ Punkt ⇒ real ⇒ Punkt"
where
"konvexpunkt S E α = S + Complex 1 (- tan (α/2)) * (E-S)/2"
```

For the computation of safety zones it not only necessary to compute over-approximations of arcs (by triangles), but these should also be as tight as possible, to keep the safety zone small. The approximation using a single convex point yields extremely large triangles for arc angles close to π, since the tangents become almost parallel. The solution is to increase the number of auxiliary points by splitting the arc into n equal segments. Fig. 7.9 is an example, if the arc from P to Q is considered as split into two halves, using the intermediate point U_1 and two auxiliary points V_1, V_2. The overall arc is then covered by the convex hull of the points P, V_1, V_2, Q. This allows to generalise the computation of the convex point to the computation of an arbitrary number $n \geq 1$ of auxiliary points. Given a start point S and an arc of length $n \cdot s$ ($n \in \mathbb{N}$) and angle $n \cdot \alpha$, the convex point of the first segment of length s and angle α can be computed. The arc transformation that defines the arc can then be used to transform this auxiliary point to obtain auxiliary points for the remaining $n-1$ segments. For example, by applying the transformation that maps P to U_1 to V_1, one obtains V_2.

As a technical aside, we note that scaling the parameters s, α of the transformation *bogenendpunkt* by a natural number n is equal to the exponentiation (iterated, n-fold application) of that function by n:

```
lemma bogenendpunkt_power:
"bogenendpunkt s α ^ n = bogenendpunkt (s*real n) (α*real n)"
```

Finally, given a start point S, an arc starting at S of length $n \cdot s$ and angle $n \cdot \alpha$ we can define the set of points whose convex hull covers the arc, hence contains all points of the arc. It consists of S, the end point of the arc (*bogenendpunkt* $(n \cdot s)$ $(n \cdot \alpha)$ S), and all transformations of the convex point of the first segment by (*bogenendpunkt* s $\alpha)^i$ $(i < n)$.

```
definition
konvexpunkt_menge :: "Punkt ⇒ real ⇒ real ⇒ nat ⇒ Punkt set"
where
"konvexpunkt_menge S s α n =
  {S, bogenendpunkt (s*real n) (α*real n) S} ∪
  {K . ∃i<n . K = ((bogenendpunkt s α)^i)
                    (konvexpunkt S (bogenendpunkt s α S) α)}"
```

Applying the convex hull operator *konvexe-huelle* to *konvexpunkt-huelle* yields the actual arc covering:

```
definition
konvexpunkt_huelle :: "Punkt ⇒ real ⇒ real ⇒ nat ⇒ Punkt set"
where
"konvexpunkt_huelle S s α n =
   konvexe_huelle (konvexpunkt_menge S s α n)"
```

The correctness of all these constructions can be summarised in a single theorem stating that the arc as defined by *bogen* is a subset of the corresponding *konvexpunkt-huelle*, hence that the latter covers the former.

```
theorem bogen_in_konvexpunkt_huelle:
"|α| < pi ⟹
   bogen (s * real n) (α * real n) S ⊆ konvexpunkt_huelle S s α n"
```

Proof (sketch). We can split an arc of length $n \cdot s$ into n parts

$$\begin{aligned}&\mathit{bogen}\ (n \cdot s)\ (n \cdot \alpha)\ S \\ &\qquad = \bigcup\{B.\ \exists i < n.\ B = (\mathit{bogenendpunkt}\ s\ \alpha)^i \text{ ‘ } (\mathit{bogen}\ s\ \alpha\ S)\}\end{aligned} \tag{7.8}$$

so that it suffices to show that for an arbitrary $i < n$

$$(\mathit{bogenendpunkt}\ s\ \alpha)^i \text{ ‘ } (\mathit{bogen}\ s\ \alpha\ S) \subseteq \mathit{konvexpunkt\text{-}huelle}\ S\ s\ \alpha\ n. \tag{7.9}$$

We have proven formally that the transformed corner points of the triangle covering the first arc segment lies in the overall proposed covering (the proof is omitted here):

$$\Delta := \{S, \mathit{konvexpunkt}\ S\ (\mathit{bogenendpunkt}\ s\ \alpha\ S)\ \alpha, \mathit{bogenendpunkt}\ s\ \alpha\ S\}$$

$$(\mathit{bogenendpunkt}\ s\ \alpha)^i \text{ ‘ } \Delta \subseteq \mathit{konvexpunkt\text{-}huelle}\ S\ s\ \alpha\ n. \tag{7.10}$$

Since the right-hand side denotes a convex set, it must also be a superset of the convex hull of the left-hand side:

$$\mathit{konvexe\text{-}huelle}\ ((\mathit{bogenendpunkt}\ s\ \alpha)^i \text{ ‘ } \Delta) \subseteq \mathit{konvexpunkt\text{-}huelle}\ S\ s\ \alpha\ n, \tag{7.11}$$

so that by transitivity with Eq. (7.9) it suffices to show

$$(\mathit{bogenendpunkt}\ s\ \alpha)^i \text{ ‘ } (\mathit{bogen}\ s\ \alpha\ S) \subseteq \mathit{konvexe\text{-}huelle}\ ((\mathit{bogenendpunkt}\ s\ \alpha)^i \text{ ‘ } \Delta). \tag{7.12}$$

We employ the fact that forming the convex hull and applying a rigid body transformation commutes, and that the image operator ‘ is monotone to reduce the proof to

$$\mathit{bogen}\ s\ \alpha\ S \subseteq \mathit{konvexe\text{-}huelle}\ \Delta, \tag{7.13}$$

which is exactly the statement that an arc is covered by the triangle construction of Fig. 7.9. □

7.3 Concrete Specifications and Verification

The purpose of this section is to highlight some applications of the update simplification rules of the memory model in concrete program code, to demonstrate that additional, tailor-made lemmas are required to prove real functions correct, and especially to show that the CSI specification language allows for readable and concise specifications of code performing geometric operations. The section does not list the details of all steps in a typical use of the verification environment. Therefore, the examples given are taken from the middle of actual verification attempts.

```
  _Bool bremskonfiguration_OK() =
    0 < mindist_bremsmessung &&
    0 < sams_konfiguration.fahrzeug.
       maximale_ausdehnung_original &&
    sams_konfiguration.bremswege.anzahl >= 2 &&
    sams_konfiguration.bremswege.anzahl <
       SAMS_BREMSDATEN__ARRSZ &&
    sams_konfiguration.bremswege.messungen
       [sams_konfiguration.bremswege.anzahl -1].v == 0.0 &&
    sams_konfiguration.bremswege.messungen
       [sams_konfiguration.bremswege.anzahl -1].s == 0.0 &&
    ${ let bs = ^BremsmessungListe{
sams_konfiguration.bremswege.messungen,
sams_konfiguration.bremswege.anzahl}
         in (sortiert-nach
               (λ(v₁, s₁) (v₂, s₂). v₁ − ‘mindist_bremsmessung ≥ v₂) bs) ∧
            sublineare-liste bs
      };
```

Figure 7.10: The specification expression bremskonfiguration_OK defines requirements on on global variables that ensure valid braking configuration settings

7.3.1 Braking Model Computations

Data Invariants

The configuration data that are required for the braking model computations are kept in global structures. This is a typical use case for *data invariants*: the global structures need to satisfy certain properties to describe a *valid* braking configuration. We use abbreviations (cf. Sec. 3.5.3) to name the expression describing their validity, as in Fig. 7.10. mindist_bremsmessung must be a positive value that denotes the minimum difference between two velocities used as braking measurements. Its concrete value is of no importance in CSI specifications, since numerical precision is not verified by the verification environment. Further subexpressions require the maximum extension of the EUC model to be positive, demand that at least two measurements exist, that the measurement array has $(0.0, 0.0)$ as its last element, that the list of braking measurements is sorted by the velocity component in descending order, with a difference of at least mindist_bremsmessung between adjacent measurements, and that this list defines a sub-linear list. The last two properties are expressed in terms of a domain representation of the array sams_konfiguration.bremswege.messungen as a list of *real* tuples.

Data invariants do not form an independent concept. They are instead emulated by adding the corresponding abbreviations to the preconditions of all functions that depend on them. This approach is more verbose than a tight integration, which might make them available automatically during verification, but keeps them implicit in specifications. On the other hand, explicitness can be regarded a benefit in safety-related specifications.

```
/*@
  @requires 0 <= v
    && v < sams_konfiguration.bremswege.messungen[0].v
    && ::bremskonfiguration_OK()
  @modifies \nothing
  @ensures 0 < \result
   && \result < sams_konfiguration.bremswege.anzahl
   && sams_konfiguration.bremswege.messungen[\result -1].v > v
   && v >= sams_konfiguration.bremswege.messungen[\result].v
  @*/
Int32 bin_suche_index_v( Float32 v );
```

Figure 7.11: Computation of the interval of braking measurements into which the given velocity v falls, via binary search

Binary Search Algorithm

Fig. 7.11 shows the specification of the auxiliary function bin_suche_index_v, which finds the upper index of an interval into which a given velocity v falls w. r. t. the braking measurement array. The specification does not employ abstraction, but instead expresses a property over C data structures. This is appropriate, since its functionality is program related (finding an index in an array). Its implementation is entirely standard[7] and is given in Sec. B.2, including the relevant invariant. A simple loop narrows two indices imin and imax down (by assigning one of them the value (imin + imax) / 2) until they meet at the interval containing v. The invariant states that v consistently lies inside the interval defined by the two indices. The proof is almost fully mechanical, which means that it consists of a mere sequence of calls to the verification tactics `pr_tac`, `pr_step`, `pr_simp`, and `pr_valid`, as well as unparameterised calls to Isabelle's simplifier via the `simp` tactic. Serious user interaction is only required to prove a couple of arithmetic goals. Concretely, the following theorems were proven during the course of verification and then added to the simplifier rule set (by applying `simp add:` $thm_1 \ldots thm_n$) in the appropriate places:

lemma *int_pos_sum_div_2:*
"⟦0 ≤ a; a + 1 < b⟧ ⟹ a < (a + b) div (2::int)"

lemma *div_2_less:*
"⟦ a ≤ b; b < c⟧ ⟹ (a + b) div (2::int) < c"

lemma *div_2_less':*
"⟦ a < b; b ≤ c⟧ ⟹ (a + b) div (2::int) < c"

lemma *div_2_nonneg:*
"⟦0 ≤ a; 0 ≤ b⟧ ⟹ 0 ≤ (a + b::int) div 2"

All these rules themselves can be proven by Isabelle's arithmetic decision procedure. (Note that modus ponens cannot be applied to the property of being automatically provable: if A and $A \longrightarrow B$ can be proven automatically,

[7] Even including the famous overflow bug for gigantic arrays that many binary searches feature, as described by J. Bloch at `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`.

```
/*@
  @requires ::bremskonfiguration_OK()
  @modifies \nothing
  @ensures
     ${ let ts = ^BremsmessungListe{
                     sams_konfiguration.bremswege.messungen,
                     sams_konfiguration.bremswege.anzahl}
        in
          ‘{\result} * |‘v| = bremsweg-approx-fkt ts |‘v|
     }
  @*/
Float32 bremsweg_geradeaus( Float32 v );
```

Figure 7.12: Computation of the *time* it takes to travel the braking distance for velocity v according to the braking model, while driving at v

then B is not necessarily provable automatically on its own.) The simple yet specialised character of these rules is common for auxiliary lemmas required during proofs with the verification environment. They are neither of general interest, so that it would be worth putting them into a library, nor are they inherently necessary to finish a proof. They are simply a means to the end of quickly finishing a current proof goal. In this case, they are needed to prove the validity of array accesses at the index (imin + imax) / 2 as well as to re-establish the loop invariant at the end of the loop body.

Straight Forward Motion

Function bremsweg_geradeaus in Fig. 7.12 is used to compute the braking distance s at straight forward motion for a given velocity v according to the braking model. For technical reasons, it returns the quotient s/v instead of s. The implementation uses the above binary search to find the measurement interval in which v lies, and computes the corresponding quotient (it is also displayed in Sec. B.2). If the maximum velocity is exceeded, the cubic over-approximation is used. Essentially, the procedure represents an efficient implementation of the domain-level function *bremsweg-approx-fkt*, a fact that is directly expressed in the postcondition, and whose proof is the main task in the overall correctness proof. Aliasing, on the other hand, poses no problems, because the function only assigns to local variables.

We only consider the normal case where a given velocity v lies inside one of the measurement intervals (i. e., effectively below the maximum velocity for which a measurement was taken). The recursive function *bremsweg-approx-fkt* can then be expressed in a closed form by using list index operations, which can later directly be related to the array index operations in the code.

```
lemma bremsweg_approx_fkt_value_normal:
  "∀i. fst (bs ! i) > v ⟶ fst (bs ! Suc i) ≤ v ⟶
    sortiert_nach (λ(v,s) (v',s'). v' < v) bs ⟶
    Suc i < length bs ⟶
      bremsweg_approx_fkt bs v =
        (snd (bs ! i) - snd (bs ! Suc i)) /
        (fst (bs ! i) - fst (bs ! Suc i)) *
        (v - (fst (bs ! Suc i))) + snd (bs ! Suc i)"
```

The precondition :: bremskonfiguration_OK provides the required property of sortedness, while the remaining three premises are satisfied by the postcondition of the binary search routine, i. e., the index it returns provides the interval bounds for v. We further point out that *bs* is instantiated with a *BremsmessungListe* term when applied to the postcondition, while the \ *result* term on its left-hand side will consist of array expressions like sams_konfiguration .bremswege .messungen[i].v. Effectively, we have to provide an operation on the domain value (the list of measurements) that corresponds to the primitive read operation in the code. This is a case where the C representation of the domain value is not used en bloc in the code, making additional lemmas necessary that equate array accesses and list indexing. For example:

```
lemma BremsmessungListe_array_acc_v:
  " ∀i m. i < n ⟶ ((m)Bremsmessung_t·[i])Bremsmessung_t→''v'' @d σ =
                      fst (BremsmessungListe_n σ m n ! i)"
```

The proof script for this function contains 19 auxiliary lemmas in total, some of which were only used in the proofs of other auxiliary lemmas, but not in the main correctness proof. The proof of function bremsweg_geradeaus is rather straightforward and consists of slightly over 200 applications of tactics. This number could seriously be reduced by a more anticipating proof strategy avoiding repetitive proof steps. However, it shows that the more abstract level of specifications entails increased manual guidance. The auxiliary lemmas shown above were not unconditionally added to the set of rewrite rules for the simplifier, but only applied in exactly the places where they were needed, preceded by a manual massaging of the proof state to make them applicable.

Computing the Braking Configuration

In Fig. 7.13 we present the function to compute the arc length s and the angle alpha from a given velocity pair v, w. Its fifth parameter t_l can be used to indicate an additional latency time during which the EUC travels unbraked. Its @memory annotation is typical: the out-parameters s and alpha have to be separate from all memory areas that are read by the function and its callees. In this case, these are the structure for error messages and logging, sams_andere, and the configuration data sams_konfiguration.

The postcondition only describes the state in the case where the return value is sams_sicher, indicating that the computation succeeded. We adopted the failure model used in the code (cf. Sec. 6.4.2) also in specifications, and only stated the specifications for successful executions of internal functions. This was sufficient, in particular since the @modifies clause, which has to hold unconditionally, excludes unwanted side-effects such as setting the variable launch_rockets to true, or at least makes their possibility visible.

The postcondition is further divided into the case where v and w are both 0, and the more interesting case where they are not. There, the equivalent straight forward velocity vG is computed via the domain function *aequiv-v-gerade*, the measurements are interpreted as a list, as before, and the overall braking time is computed using *bremsweg-approx-fkt* and the latency parameter t_l, as well as the pre-configured latency (sams_konfiguration . einstellungen .t_r). Finally, the output values stored in *s and *alpha are specified, and vG is guaranteed

```
/*@
  @requires ::bremskonfiguration_OK()
    && \valid(s) && \valid(alpha)
  @memory *s <*> *alpha <*> (sams_andere <|>
                              sams_konfiguration)
  @modifies sams_andere, *alpha, *s
  @ensures \result == sams_sicher -->
     ((v == 0 && w == 0 --> *s == 0 && *alpha == 0) &&
     (v != 0 || w != 0 -->
       ${
        let vG = ‘{$aequiv_v_gerade(v, w,
sams_konfiguration.fahrzeug.maximale_ausdehnung_original)};
            ts = ^BremsmessungListe{
sams_konfiguration.bremswege.messungen,
sams_konfiguration.bremswege.anzahl};
            t = bremsweg-approx-fkt ts vG / vG +
‘{sams_konfiguration.einstellungen.t_r + t_l}
        in
           ‘{*s} = ‘v * t ∧ ‘{*alpha} = ‘w * t ∧
           vG ≤ ‘{sams_konfiguration.fahrzeug.v_max}
        }))
  @*/
SAMSStatus berechne_bremskonfiguration( Float32 v,
                                        Float32 w,
                                        Laenge *s,
                                        WinkelRad *alpha,
                                        Float32 t_l );
```

Figure 7.13: Computation of (s, α) for a given (v, ω) and latency t_l

to be bounded by the configured maximum velocity (or else sams_sicher would not have been returned).

The proof of this function is much simpler than the previous one, since the relevant domain abstraction work is done by the called functions, i. e., bremsweg_geradeaus itself uses all C representations en bloc, and performs no complicated arithmetic that could not be handled by Isabelle's simplifier. Only two trivial auxiliary lemmas were needed, and the proof consists of less than 50 tactic applications.

7.3.2 Arc Coverings

Function bogenhuelle_L in Fig. 7.14 implements the approximation of arcs given by their length s and angle alpha using L auxiliary points, as described on the domain level in Sec. 7.2.2. For practical reasons, the function does not approximate a single arc, but yields approximations of all arcs described by the points in the given array startpunkte_daten, which in practice constitutes the contour points of the EUC.

A successful execution of this function conceptually proceeds as follows. The relevant array lengths are ensured (i. e., the correct relation between startpunkte_laenge and ergebnis_laenge_max is checked) and L >= 1 is true. Then, for each of the EUC contour points, the corresponding arc is split into L segments of equal

```
/*@
  @requires
     startpunkte_laenge * L <= ergebnis_laenge_max
   && 4 < L
   && $fabs(alpha) <= 2 * sams_pi
  @memory startpunkte_daten[:startpunkte_laenge] <*>
          ergebnis_daten[:ergebnis_laenge_max] <*>
          sams_andere
  @modifies
    sams_andere, ergebnis_daten[:startpunkte_laenge * L]
  @ensures (\result == sams_sicher) -->
    ${ ∀ i. 0 ≤ i ∧ i < ‘startpunkte_laenge ⟶
         let sp = ^Vektor2DR{&startpunkte_daten[$i]};
             X = {sp, bogenendpunkt ‘s ‘alpha sp} ∪
                    {q. (∃ j. 0 ≤ j ∧ j < ‘L ∧
                           q = ^Vektor2DR{&ergebnis_daten[$i + $j
                                         * startpunkte_laenge]})}
         in (bogen ‘s ‘alpha sp) ⊆ konvexe-huelle X)
    }
  @*/
SAMSStatus bogenhuelle_L( Laenge s,
                          WinkelRad alpha,
                          Int32 L,
                          const Vektor2D * startpunkte_daten,
                          Int32 startpunkte_laenge,
                          Vektor2D * ergebnis_daten,
                          Int32 ergebnis_laenge_max );
```

Figure 7.14: Function computing an arc approximation using L auxiliary points

length, and the endpoint of the first segment (given by s / L and alpha / L as arc transformation parameters) is computed. The start- and endpoints of this segment are passed to a subroutine (bogenhuelle_1) that computes the convex point for the segment. This convex point is subsequently arc-transformed to obtain the convex points for the remaining L − 1 segments, so that the approximation depicted in Fig. 7.9 is achieved. All convex points thus computed are stored in the result array ergebnis_daten.

The postcondition states that for all points *sp* in the input array, the arc described by this point (*bogen s alpha sp*) is covered by the convex hull of the corresponding point set X consisting of *sp*, the arc endpoint *bogenendpunkt s alpha sp* and the associated convex points stored in the output array. This is, quite notably, a high-level property expressed on the domain level, and not merely a statement about the values of C data structures. Only in one technical detail, the specification tells for each individual contour point at which indices its convex points are stored.[8]

The correctness proof has a clear conceptual structure. For an arbitrary point *sp* of the input array, it must be shown that its arc is covered by the convex

[8] With hindsight, it would have been beneficial during verification to make coverings of multiple arcs an independent domain concept, so that the point arrays could have been treated more uniformly, avoiding arithmetic in indices and the need for several auxiliary lemmas.

hull of X. We instantiate theorem *bogen-in-konvexpunkt-huelle* as following, writing α for the function parameter *alpha* for brevity:

$$\begin{aligned}&\mathit{bogen}\ ((s/L) * \mathit{real}\ L)\ ((\alpha/L) * \mathit{real}\ L)\ \mathit{sp} \\ &\qquad\subseteq \mathit{konvexpunkt\text{-}huelle}\ \mathit{sp}\ (s/L)\ (\alpha/L)\ L\end{aligned} \tag{7.14}$$

so that by transitivity the proof goal becomes

$$\mathit{konvexpunkt\text{-}huelle}\ \mathit{sp}\ (s/L)\ (\alpha/L)\ L \subseteq \mathit{konvexe\text{-}huelle}\ X \tag{7.15}$$

which can further be reduced due to monotonicity of *konvexe-huelle*

$$\mathit{konvexpunkt\text{-}menge}\ \mathit{sp}\ (s/L)\ (\alpha/L)\ L \subseteq X. \tag{7.16}$$

We simplify further, along the definitions of X and *konvexpunkt-menge*.[9]

$$\begin{aligned}&\{K.\ \exists i < L.\ K = (\mathit{bogenendpunkt}\ (s/L)\ (\alpha/L))^{i} \\ &\qquad\qquad (\mathit{konvexpunkt}\ \mathit{sp}\ (\mathit{bogenendpunkt}\ (s/L)\ (\alpha/L)\ \mathit{sp})\ \alpha)\} \\ &\subseteq \{q.\ (\exists j.\ 0 \leq j \wedge j < L \wedge \\ &\qquad q = \mathit{Vector2DR}\ \sigma\ (\mathit{ergebnis\text{-}daten}_t.[i + j * \mathit{startpunkte\text{-}laenge}]))\}\end{aligned} \tag{7.17}$$

The stated subset relationship actually is a proper equality, which is proven by showing, for some $i' < L$

$$\begin{aligned}&(\mathit{bogenendpunkt}\ (s/L)\ (\alpha/L))^{i'} \\ &\qquad (\mathit{konvexpunkt}\ \mathit{sp}\ (\mathit{bogenendpunkt}\ (s/L)\ (\alpha/L)\ \mathit{sp})\ \alpha) = \\ &\qquad \mathit{Vector2DR}\ \sigma\ (\mathit{ergebnis\text{-}daten}_t.[i + i' * \mathit{startpunkte\text{-}laenge}]).\end{aligned} \tag{7.18}$$

This equation is simply the statement that the output array contains, at the given index, the convex point of the i'-th segment of the arc described by *sp*, obtained by transforming the convex point of the initial segment. But this is exactly what is achieved by the abovementioned algorithm.

So far, the proof sketch has ignored the necessary update simplification steps. For example, the state σ in Eq. (7.18) will be the post-state of the function, so that update simplification must be applied to compute its actual value. Unfortunately it is necessary to manually insert array bounds assertions and apply congruence rules at several places in the proof, which obfuscates its structure. Concretely, to simplify the vector term in Eq. (7.17), we (manually) apply the following tailor-made congruence rule:[10]

```
lemma Vector2DR_set_index_cong[rule_format]:
"(∀j. 0 ≤ j ⟶ j < n ⟶ ϱ(vec) σ' (l j) = ϱ(vec) σ (l j)) ⟹
 {q. ∃(j::int) ≥ 0. j < n ∧ P q j (ϱ(vec) σ' (l j))} =
 {q. ∃j ≥ 0. j < n ∧ P q j (ϱ(vec)  σ (l j))}"
```

This rule allows us to rewrite a representation function term that includes a location l which depends on an existentially bound index variable j while assuming that j lies within its specified bounds. This is necessary to be able to apply rules such as *Vec2DRec-n-update-other* (cf. Sec. 4.5.2), which are conditional over array bounds.

[9] The 'free' variable i on the right-hand side is the index of *sp* in the input array.

[10] Recall that $\mathit{Vector2DR}\ \sigma\ l = \tau(\mathit{vec})\ (\varrho(\mathit{vec})\ \sigma\ l)$, where $\varrho(\mathit{vec})$ yields an intermediary Isabelle record and $\tau(\mathit{vec})$ transforms it into the final domain value (cf. Sec. 4.5.1)

At one point, it was necessary to prove that an array index a is valid, under the assumption that the index $a * n$ ($n > 0$) was valid:

```
lemma left_mult_le_cancel:
"⟦0 ≤ a; a * n ≤ b; (0::int) < n⟧ ⟹ a ≤ b"
```

The Isabelle/HOL library contains several of similar lemmas, but none of them allowed the proof procedure invoked during update simplification to prove the validity of the array access, so that the above lemma had to manually be proven and added to the procedure's rule set.

Another situation where auxiliary lemmas were required was related to the scaling of s and alpha by fractions of L. The prover was not able to derive that if $|\alpha| \leq 2\pi$, then a scaling of α by a particular factor between 0 and 1 is, too. So it had to be tuned by adding the following lemma, where the specific term structure in the conclusion matches that which appeared in the proof state:

```
lemma aux_alpha_k_over_l:
"⟦ 1 ≤ k; k < l; abs α ≤ 2 * pi ⟧ ⟹
  abs (α * k / l - α / l) ≤ 2 * pi"
```

In total, 19 auxiliary lemmas were deemed necessary during the proof of function bogenhuelle_L. The correctness proof consists of slightly over 300 applications of tactics.

7.4 Reflection

This section is concerned with a (subjective) evaluation of the verification environment. It is based on the experiences made while formally verifying functional properties of the SAMS software with it. In particular, we highlight the errors that were found thanks to its use as well as the effect it had on the overall verification process. The limitations regarding both its scope and its technical realisation are also pointed out.

7.4.1 Key Figures

While we have concentrated on selected functions and their verification above, we provide here some general figures about the software and the SAMS project.

The project staff comprised seven persons according to the project plan. The duration of the project was three years. This time includes the initial phase during which the product requirements were worked out. The verification environment was designed and built during this time, too. This makes it hard to tell how much time went into its design and development and how much time went into the verification of the software functions. We found shortcomings in the tool's realisation, particularly efficiency problems when dealing with large functions, even while we verified functions from the SAMS software.

The final version of the software module computing safety zones was handed to the certification authority in September 2009. The correctness proofs of the functions that comprise this module were done by the author in the last six months of the project. Similar functions had been verified before. However, no substantial parts of the individual proof scripts for functions from the SAMS software existed before.

The software module consists 11 C files and 15 C header files at an overall size of 240 kB. The total number of lines of code in these files is 2804. Comments and specifications (which syntactically are comments, too) add up to 2535 lines. (These figures were determined using the `cloc` tool[11].) The C files contain definitions of 39 functions in total, ignoring functions for logging purposes and to record control flow. All of these were formally specified and both the specifications and the source code were examined in specification review sessions. The largest specification with 63 lines was that of the top-level function to compute the safety zone. Its size is substantial, but does not exceed the limits for a comprehensible specification in our opinion. The structuring of specifications with the help of `@abbreviation` annotations proved crucial to keep specifications readable. Eventually, 29 functions (74%) were formally verified when the software was handed to the authority.

Two functionally interesting, but rather long functions were not verified. They are a function implementing the well-known Graham Scan algorithm [120] to compute the convex hull of a set of points as well as the function that samples the safety zone into a laser-scan like representation (Step 3 in Sec. 7.1.2). With its 149 lines of code, a verification of the latter function was simply not manageable. The number of assumptions in the proof state became so large that our own update simplification, but also Isabelle's simplifier ran for minutes without visible progress.

Related efforts targeting the verification of realistic C programs include the full functional verification of the seL4 microkernel [89]. The verified software comprises 8700 lines of C code and 600 lines of assembler. They report a total verification effort of 20 person years (py), including time to develop the necessary verification infrastructure and including 11 py for the correctness proof. The latter is divided into a refinement proof from an abstract (non-deterministic) specification to a concrete one, which took 8 py, and the proof relating the concrete specification to the source code (3 py). Another example is the verification of a non-optimising compiler for a subset of C called C0, which itself is written in C0, by Leinenbach and Petrova [95]. The program consists of 1500 lines of C0 code whose verification took about one person year. Our work compares favourably to these if one considers the combination of code size and verification effort.

7.4.2 Errors Found

Myers et al. [112] define testing as the process of executing a program with the intent of finding errors. Dijkstra, with a more critical attitude towards testing, argues that testing can be used to show the presence of bugs, but never to show their absence. Put so generally, this statement is certainly not true: exhaustive testing can provide the same strong guarantees as a formal correctness proof. Moreover, formal approaches to testing demonstrate that there is no insurmountable border between formal methods and testing [125, 124, 30, 37]. Nevertheless, formal methods are often contrasted to testing by arguing that they are inherently able to prove the complete absence of certain classes of bugs from a given program. Yet the number and kind of found bugs, or errors, is also a valid criterion for evaluating the use of a formal verification

[11] Freely available at `http://cloc.sourceforge.net`. The version used was 1.51.

environment. In every non-trivial program there will initially be deviations from the specification, or the specification itself will be flawed or incomplete. To find these flaws and eradicate them is one crucial step in achieving safety in software. We consider it as important as asserting that certain classes of errors are not present in the end product, because this final step cannot be reached without the preceding iterative process of improving both specification and software. A useful verification tool should therefore help discovering flaws in the software.

But the use of a tool or methodology —in particular one that has a large influence on the overall development process— can also have 'hidden' effects on software quality. We believe that this was the case for the use of the formal verification environment in the SAMS project. One reason is that the code was continually implemented with ease of verifiability in mind. Code quality increases if the developers are involved in creating sketches of the correctness proofs for central program functions. Obviously, this calls for highly competent and mathematically inclined developers. A further reason is that the verifiers and domain experts took part in code reviews and could add their expertise about potential problems in the code. This led to an ultimately rather low number of errors that were discovered during the actual function correctness proofs. This observation is also reported by Klein et al. [90], who point out that in their formal verification of an operating system microkernel, *no deeper algorithmic bugs* were found during source code verification, as these had already been detected or avoided while working out the low-level specification. The following three sections list some of the errors we found .

Errors Uncovered During Domain Modelling

High-level errors were uncovered by virtue of the formal domain modelling. For this, no associated code needs to exist, even if in the cases at hand it did (as explained in Sec. 7.4.3 below and in [61]). For example, the initial definition of the function *bremsweg-approx-fkt* modelling the EUC's braking behaviour was written under the assumption that a quadratic over-approximation of the braking distance above the maximum measured velocity would suffice to obtain a convex function (cf. theorem *bremsweg-approx-fkt-konvex* on page 158). However, a formal proof attempt of this fact unveiled that, in fact, a cubic approximation would be necessary. Both code and specification were amended to incorporate this insight.

This concrete error would probably not have caused failures in the deployed system; however, the error was structurally a faulty assumption about the application domain. Such errors are hard to detect in general. Assessing domain assumptions by formalising them and trying to derive their 'obvious' properties is an effective means for tackling this problem.

Errors in the Specification

Formal verification is extremely well-suited for ensuring the completeness of specifications, i. e., to uncover hidden assumptions or to highlight omissions. While formally verifying the code, several minor flaws were detected, such as asserting that the variable keeping the maximum velocity is strictly positive, or that the configured number of laser beams is below a threshold. Since array lengths and array pointers must be maintained separately in C, several preconditions of the kind

```
\array(a, len) && 0 <= idx && idx < len
```

were used, some of which were missed until they were needed during verification.

A more serious bug was found in the specification of the top-level routine computing the safety zone: it described the function's output as the intersection of the monitored area (the area visible to the laser scanner) and the computed safety zone. However, both areas were expressed in different coordinate systems; the former took the position of the laser scanner as its origin, while the latter used the reference point of the EUC (depicted in Fig. 7.7). The obvious solution was to apply a coordinate transformation to the safety zone before intersecting the two. The bug was discovered while planning the correctness proof of the function on paper. In fact, the bug was found because the transformation was applied in the code from the beginning, leading to a mismatch with the specification.

Errors in the Code

There were relatively few errors in the SAMS code that could be detected through formal verification. We attribute this fact to the thorough scientific scrutiny under which the domain model, the specifications as well as the code were put at virtually all times. Moreover, SAMS being a research project, there was certainly less pressure w. r. t. time to market and cost efficiency, which commonly benefits quality. All persons involved in the specification, design, development and verification of the safety-related software collaborated closely. This situation is probably uncommon for a purely industrial project, where the pressures are stronger and where project members might be separated across several companies. We therefore assume that the number of errors found in such projects would be greater than it was in the SAMS project. Of course one can argue that time to market pressure would in fact prevent the application of time-consuming verification methods in the first place. There are however indications that formal specification and verification actually increase cost-effectiveness [11].

Nonetheless, the following two errors were actually only found during formal program verification. In function bogenhuelle_L we found a 'disguised' off-by-one error: the transformation that was applied to the convex point of the first segment to obtain the remaining convex points was implemented as

```
s_1     = s / (Float32)l;
alpha_1 = alpha / (Float32)l;
for (k = 1; k < L && ret == sams_sicher; ++k)
  /* ... */
  bremskonfiguration_zu_skt( s_k - s_1,
                             alpha_k - alpha_1,
                             &bogen_k );
```

where bogen_k was supposed to receive the desired transformation matrix for the $(k+1)$-th segment in an iteration. We discovered that the subtraction of s_1 and alpha_1 resulted in the duplicate computation of the convex point for the first segment, but left the final convex point uncomputed. The error was fixed, and verification succeeded.

The other error was discovered in a function performing the standard geometric test whether a point lies mathematically to the left of a given straight line, by computing a determinant and checking its sign (endpunkt_links_von_richtung). The test was supposed to check for strict left-ness, i. e., it should reject points lying on the line. Since it used a (<) comparison instead of (<=), the behaviour was incorrect.

7.4.3 Verification Process

In this subsection we reiterate the arguments made in [150] about the characteristics and benefits of the verification process adopted in the SAMS project.

Verification as a joint effort. One aspect of formal verification is that because correctness relies on formal proof, it is not that crucial anymore to strictly separate the roles of tester/verifier and implementer. In contrast, the close cooperation between the verifier and the implementer boosted productivity in our case: verification became a joint effort. Writing specifications which validate the safety requirements, and can be formally verified, is not easy; it requires an understanding of the implementation, the domain model, and how the verification works. It is easy to specify something which is correct but cannot be verified; on the other hand, it is also a temptation to write low-level specifications which just restate what the code is doing in elementary terms without the abstraction required to state useful safety properties. In our case, the specifications were authored together in regular specification meetings. Sometimes a specification was only formulated after prototypical code had been written and initially reviewed for obvious errors. These meetings ensured a good understanding of the specifications by the implementers, and vice versa a sufficient understanding of the code by the verifiers to enable them to do their work.

A somewhat unusual example of a close collaboration between implementer and verifier is a change of the implementation induced by verifiability considerations. It refers to the function to convert the safety zone into a sequence of vectors corresponding to a laser scan (Step (e) in Fig. 1.2). Initially, the specification interpreted the resulting sequence as the rays of an idealised laser scanner. We switched both specification and implementation to a sector-based interpretation, in which each result describes the whole area of a sector. This fitted in well with the other specifications and allowed us to specify the result simply as a superset of the actual safety zone, and was easier to verify formally. Again, we share this observation with Klein et al. [90], who regard it essential to be able to modify the source code with the aim of easing verification to complete the verification on time.

In personal communication —among others with members of CEA-LIST[12], who were at that time cooperating with Airbus to formally verify C code— it was often noted that industrial partners are very reluctant when it comes to giving away their source code, even to project partners, let alone accepting proposals for code changes. Therefore, the situation in the SAMS project was probably exceptionally well-suited for a formal verification effort.

[12] Software Reliability Laboratory of the Commissariat à l'Energie Atomique, Saclay, France.

Code-centric specification and verification. We experienced another interesting interplay between specification, implementation and application. It shows that in the application domain (safety-related robotics software) a strict waterfall-like development process (of which the V-model is a descendant) is not suitable or at least entails problems. This is because apart from safety, the availability of systems is also of sublime importance, but can really only be assessed on running systems. This leads to a feedback loop from the lowest parts of the model —the coding level— up to the specification and design levels. The argument is independent of the application of formal methods, but noteworthy nonetheless. At first, the specification required that an emergency stop should be initiated whenever the speed of the vehicle exceeded the maximum speed for which a braking distance was measured. However, simulations on a prototypical implementation revealed that this requirement was too restrictive: in typical applications, the measured maximum velocity v_m may be exceeded occasionally by a small margin, and initiating an emergency stop in these situations would severely reduce availability. Hence, the requirement was modified so that the braking distance for speeds larger than v_m could safely be over-approximated, and the specification adapted accordingly.

The importance of being formal. Formal specification necessitates to state requirements precisely. A beneficial side effect is that it focuses discussions and manifests design decisions. Besides the well-known issue of the ambiguities in natural language specifications, it turned out to be easier for specifiers *and* implementers to use the vocabulary of the domain formalisation to state these requirements and to reach agreement on their respective meaning. For quick sanity checks of specifications written down or modified during meetings, we provide tool support for the type-checking of specifications. This pertains both to code-related specification expressions (e. g., types of program variables) as well as Isabelle expressions used in code specifications. A typical specification meeting would end with a function specification reviewed and type-checked.

7.4.4 Impact of Changes

A major annoyance is the fragility of proofs, i. e. their lack of robustness w. r. t. changes in source code. This particularly hurts in the face of interactive verification: proofs are not generated automatically by a push-button tool, but proofs scripts are written by humans —even if they sometimes only consist of a short sequence of calls to automatic proof tactics. We easily support 'regression verification', i. e. the automatic checking of all existing correctness proofs against modified source code as well as modified specifications.[13] Unfortunately, however, many proof scripts become invalid even through small modifications like the rearrangement of statements or a semantics-preserving rewriting of expressions. This meant that they had to be adapted manually in most cases.

This problem only stays tractable because the change impact is always shielded at the function interface boundaries: As long as the interface specification of a function f does not change, all its callers are not affected by changes in the body of f. If the specification is modified, all callers need to be

[13] This functionality is not built into the verification environment itself, but is instead realised by a collection of scripts that are executed once a day, or on demand.

re-verified; again, *their* callers are not affected, as long as no changes to other specifications become necessary.

7.4.5 Technical Realisation

By technical realisation we refer to the issues pertaining to the use of Isabelle as the prover back-end and interface for the verifier, and more specifically the use of higher-order logic, i. e., Isabelle/HOL, for all parts of the formalisation. We do not intend to argue for or against the use of either, which has been done elsewhere [122, 105, 5, 88, 100]; instead, we want to point out two aspects that emerged during our practical verification work.

Structured proof states. A lot of information has to be maintained during a correctness proof, ranging from the specification of the function to be proven over a symbolic encoding of one or several program executions to intermediate facts such as the results of update simplification steps, which may be kept for later re-use. Unfortunately, provers like Isabelle represent proof states in a very simplistic structure essentially consisting of a list of assumptions and a single conclusion. This forces one to encode a more complex structure on the level of the object logic. For example, we introduced another equality operator ($\doteq$) to express relations between program states (e. g., $\sigma' \doteq \sigma(l ::= v)$) and identified it with the regular equality (i. e., $x \doteq y$ iff $x = y$). This was done simply to be able to recognise those equalities relevant to update simplification. Having two operators for equality is irritating from the user's point of view. Moreover, having to maintain such equalities in the user visible proof state clutters up the proof state display, while they are in most cases not of interest to the user.

The same applies to caching the results or by-products of expensive proof steps. For example, we implemented a procedure performing update simplification on an arbitrary expression containing read terms ($l \mathbin{@_i} \sigma$, etc.). In addition to substituting the final results of all update simplifications, the procedure adds the associated equations (e. g., $l \mathbin{@_i} \sigma \doteq v$) as assumptions and re-uses them subsequently. This speeds up the whole update simplification process by an order of magnitude, but adds several more technical assumptions.

The efficiency of proof procedures is furthermore impaired by the flat assumption structure, because it often depends on the number of facts (and thereby assumptions) to consider. This is true in particular for Isabelle's simplifier, which can easily get trapped in an infinite rewrite loop, whose cause is really hard to detect in the face of over a hundred assumptions and rather poor support for tracing its steps. We invented dummy marker predicates with trivial definitions, such as $\mathit{INV}\ x = x$ for loop invariants, and fine-tuned the congruence rules of the simplifier to prevent it from considering the marker's argument during its operation. In the case of *INV*, the latter is a user-defined invariant expression and can hence include a subexpression evoking non-termination.

Fine-grained control over proof steps. Another stumbling block is the restricted possibility of controlling individual proof steps. A characteristic example is the proof that a given array index lies within certain bounds. Sometimes Isabelle's powerful arithmetic proof procedure is able to find a proof, but only when given the right (and often rather obvious) selection of assumptions manually, while it does not terminate when given all assumptions of the proof state. The matured user interface provides no help in performing this selection, and Isabelle's traditional proof style is not geared towards user-guided assumption management either. The KeY prover for program verification [19] has a richer user interface and lets the user select subterms to which so-called taclets (reminiscent of Isabelle's tactics) are to be applied.

We considered switching to the Isar structured proof language by Wenzel [152] (cf. Sec. 2.2.1). However, its philosophy of making all intermediate facts and assumptions literally explicit in the proof script quickly became inconvenient. A user does not want to (and for efficiency reasons must not be required to) type in the large number of technical assumptions that occur during program verification. In contrast to proofs on the domain level, which are often mathematically pleasing and structurally concise, proofs of programs consist largely of long sequences of technical steps.

7.4.6 Limitations

Our tool focuses on functional correctness, and does not consider aspects like execution time analysis and bounds, resource consumption, concurrency, and the interface between hardware and software. This is a clear separation of concerns, as it is becoming common consensus that only the use of multiple, specialised tools and methodologies can achieve a high level of confidence in software [78]. There are further limitations in the realm of functional properties and run-time errors. Like other formalisations, we idealise the numerical domains that programs work on from bounded integers and floating-point numbers to mathematical integers and real numbers. This may in exceptional cases result in undetected run-time errors; we have given an example in [100]. The price we had to pay to obtain a formalisation in which interesting, abstract, functional properties can be proven with tolerable effort was a slight mismatch between the actual (machine-) and the formal semantics.

7.5 IEC 61508 Safety Process Integration

Two questions arise when a tool such as our verification environment is used in an IEC 61508-3 software development. Firstly, which requirements are inflicted on the tool itself to qualify its use? And secondly, what is the benefit of the tool's usage, primarily w. r. t. the fulfilment of regulatory stipulations regarding the verification and validation of the developed software?

7.5.1 Tool Qualification

The notion of tool qualification or tool certification is present in several safety standards, but with differing degrees of detail. They have in common that the high level of rigour with which safety-related systems are developed to some degree permeates the tools used for its development.

DO-178B

DO-178B defines tool qualification as *"the process necessary to obtain certification credit for a software tool within the context of a specific airborne system."*

Certification credit is in turn defined as *"acceptance by the certification authority that a process, product or demonstration satisfies a certification requirement."* So whenever a requirement of the standard shall be satisfied by the use of a tool whose output is not individually verified, the tool must be qualified (§ 12.2). Tools can only be qualified on a per project basis, although a previous qualification obviously supports the qualification for similar projects. This generates incentives for the creators of verification and validation tools to make them "qualifiable", or to provide reference qualifications, because their users are formally guaranteed to profit from the certification credit. Kornecki and Zalewski [92] provide an overview of tool qualification in the development of dependable software that is based on DO-178B.

IEC 61508

IEC 61508 highly recommends the use of *certified tools* (Part 3, § 7.4.4 and Table A.3) from SIL 2, *"whenever possible"* (Part 7, § C.4.3). However, it does not introduce any criteria by which one can decide whether a given tool has to be certified, nor does it list requirements for a tool certification, such as the documentation of use cases or validation measures for the tool. The amount of information about tool certification is rather sparse overall. IEC 61508-3 makes the developer of the safety-related software (as opposed to the tool vendor) responsible for demonstrating that the applied tools are compliant with the requirements of the standard. The effect is similar to the requirement of DO-178B which explicitly puts tool qualification on a per project basis.

The standard provides the alternative of using tools with increased confidence from use, and highly recommends them for all SILs. In combination with the fact that the standard is agnostic of any kind of certification credit, this limits the motivation for tool developers to perform tool certification for IEC 61508-3.[14] Since little guidance is provided on how to certify a tool, most certifications in practice are individual efforts arising from a collaboration between the tool vendor and a certification authority [64, 46].

ISO/DIS 26262

The descendant of IEC 61508 for the automotive domain, ISO/DIS 26262 *Road vehicles — Functional safety*, pays more attention to the role of tools in the safety life cycle. It introduces the concepts of tool impact, tool error detection and tool confidence level. *Tool impact* is defined as the possibility that a safety requirement is violated by a failure or an erroneous output of a software tool, while *tool error detection* describes the probability that a failure or an erroneous output of a software tool will be detected (by other verification measures). The *tool confidence level* is finally a discrete measure of confidence in the tool computed from the previous two — ranging from TCL1, where no qualification is necessary, to TCL4 requiring extensive qualification.

Four methods are recommended to achieve a qualification: (1) increased confidence from use, (2) evaluation of the development process, (3) validation of the software tool, and (4) development in compliance with a safety standard.

[14] We learned in personal communication with the Institut für Arbeitsschutz der Deutschen Gesetzlichen Unfallversicherung (IFA) that this situation is about to be improved in the upcoming second edition of the standard.

Conrad et al. [47] describe how a code generator and a static analysis tool have been qualified as compliant with this standard using the third method.

7.5.2 Certification of the Verification Environment

Since the verification environment was only developed during the course of the SAMS project, invoking increased confidence from use was not an option to justify its employment. This made a tool certification necessary. To this end we cooperated with the certification authority TÜV Süd Rail GmbH. The certification effort was divided into the following work packages:

Defining the scope of formal verification This entails extracting from the standard which requirements are supposed to be fulfilled by the application of the verification environment. A straightforward mapping from requirements to the functionality of the tool could be achieved based on the tables listing procedures and measures in Annexes A and B of IEC 61508-3. The mapping is described subsequently in Sec. 7.5.3.

Moreover, we used the verification and validation plan to delineate the formal verification from other verification tasks. For the life cycle phases of module design and coding we used the following five verification methods:

- Code reviews
- Formal specification of the program functions
- Reviews of formal specifications
- Formal proof in the verification environment
- Dynamical analysis and test; these were used to demonstrate that no overflow or underflow of floating point expressions occurs and that the maximum rounding error lies within specified bounds. The RT-Tester tool [148] was used for this purpose.

It is noteworthy that we defined the formal specification of program functions as a verification activity as well as a specification activity. Very generally, the reason is that by structuring and implementing code with formal verification in mind, the code quality is already improved. More concretely, by writing formal specifications for program functions which are formulated in terms of higher level domain concepts we ensure both that the domain concepts are sufficient to express these specifications, and that the code's module structure is suitable to satisfy the requirements dictated by the domain modelling.

Modular verification on a function-by-function basis allowed us to focus formal verification on those functions which are crucial to functional correctness; other functions may contain constructs that our tool cannot reason about, or may not pertain to global correctness (e. g., logging), and can be treated more adequately by manual review or functional tests.

Providing documentation The following documentation about the verification environment and its planned use for the verification of the SAMS safety-related software were handed to the certification authority:

- A set of slides from a presentation demonstrating the approach and the scope of the verification environment, which was held in front of the certification authority.
- A plan for validating compliance with the MISRA C guidelines. Most of the rules set forth therein are checked automatically by the front-end. Some rules are not easily checked automatically, such as the requirement that *"limited dependence should be placed on C's operator precedence rules in expresions"* [109, Rule 12.1], or that there shall be no unreachable code. The implementation of the MISRA analyser is described further in the master's thesis by Märtins [103]
- A validation plan for the verification environment itself. The plan describes Isabelle's LCF-style architecture (cf. Sec. 2.2) which allows one to restrict the correctness analysis to a well-defined small kernel. It also includes the calculation that justifies Isabelle's increased confidence from use. It finally describes validation measures for the parser component of the front-end, for the formalised semantics as well as for the notion of function correctness as presented in Sec. 6.2.1. We appeal to internal reviews as well as reviews of our publications by the scientific community as the validation measures for the latter two.
- The abovementioned verification and validation plan
- A reference manual of the verification environment describing its functionality and usage
- Publicly available course material for the Isabelle theorem prover[15]. The material is divided into four sessions and consists of 132 presentation slides and 15 Isabelle/HOL example theories

Inspection and audits An on-site audit was done by the certification authority in which the verification of several example programs as well as concrete functions from the SAMS software were demonstrated. Moreover, reviews of all documents from the previous work package were performed.

Issue of letter of conformance and technical report Based on the on-site audits and the reviews of the verification and validation plan in particular, the software verification and validation procedures were acknowledged to fulfil the SIL 3 requirements according to IEC 61508-3. This was confirmed in a letter of conformance as well as a more detailed technical report, from which we quote the following: *"*SAMS verification environment *and the theorem prover* Isabelle *are applicable in use for the specification and formal verification of MISRA-C software libraries according to the standard IEC 61508-3:1998 up to SIL 3. [. . .] The related analyses and tests have shown that there are no safety-related objections against the use of* SAMS verification environment *for software verification in the phases system design, module design and implementation."*

Formal proof was recognised as the central means of verification. In contrast to the new extensions of Isabelle/HOL in the form of theories and tactics that lie at the heart of the verification environment, we successfully argued that

[15] Available at `http://isabelle.in.tum.de/coursematerial/IJCAR04`

the bare Isabelle system has already acquired increased confidence from use: *"Isabelle has been used by scientists and mathematicians in academia. Hence, it is considered a proven tool for the formal verification."* In fact, we estimated the number of hours that Isabelle has been in serious use as $2 \cdot 10^6$ hours. The technique of demonstrating increased confidence from use by estimating usage hours, by evaluating the tool's development activity, by ensuring a proper attention to defect reports and by securing the public maintenance of a list of known defects is commonly applied for non-certified compilers and was adopted here.

Open-minded authorities.

To our surprise the external reviewers from the certification authority were quite open-minded towards the use of expressive (higher-order) formal logic for specifications and an interactive theorem prover for doing the actual verification. In our case this was Isabelle/HOL, but its specifics did not play an important role and HOL4 or Coq or any other well-known prover with an active research community, ample documentation and a large enough number of global usage hours could have been chosen.

7.5.3 Covered Verification Measures

Annex A of IEC 61508-3 gives guidance on the selection of procedures and measures to fulfil the requirements of the different life cycle phases defined in the standard. It is presented in the form of ten tables, whose headings we list here for reference: (1) Specification of software safety requirements, (2) Software design and development: software architecture, (3) Software design and development: tools and programming languages, (4) Software design and development: detailed design, (5) Software design and development: software module and integration testing, (6) Integration of the programmable electronic, (7) Software safety validation, (8) Modification, (9) Software verification, (10) Functional safety assessment. The individual measures are detailed in associated tables in Annex B.

Procedures and measures that can arguably be covered by using the verification environment belong to tables A.1, A.2, A.4, A.5 and A.9. Regarding Table A.1, the standard unconditionally demands that the safety requirements are primarily specified in natural language amended with a mathematical description if applicable. It asks for the use of semi-formal or formal methods as an additional technique to increase the clarity of the specification. This requirement is obviously satisfied by formalising requirements in terms of the formal domain model in the verification environment. Table A.2 mainly asks for techniques that increase error detection and fault tolerance. Among others, it also asks for the use of formal methods. The less obvious measures that are covered have been explicitly enumerated by the certification authority in the letter of conformance. The list is reproduced in Fig. 7.15.

With regard to Table A.4, four out of six measures are covered: the use of formal methods, of computer-aided design tools, of design and coding guidelines, and of structured programming. Missing are defensive programming and modularisation. The standard interprets modularisation structurally, and our tool does not apply code metrics. In practice, functions which can be formally

Table A.4	Software design and development: detailed design	
	1c	Formal methods such as, e. g., CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM, and Z
	2	Computer-aided design tools
	5	Design and coding guidelines (detailed in B.1)
	6	Structured programming
Table A.9	Software verification	
	1	Formal proof
	3	Static analysis (detailed in B.8)
Table B.1	Design and coding guidelines	
	1	Use of coding guidelines
	2	No dynamic objects
	3a	No dynamic variables
	4	Restricted use of interrupts
	5	Restricted use of pointers
	6	Restricted use of recursions
	7	No unconditional jumps in programs written in higher-level languages
Table B.8	Static analysis	
	1	Marginal value analysis
	3	Control flow analysis
	4	Data flow analysis
	5	Error estimation
	8	Symbolic execution

Figure 7.15: Covered measures required by IEC 61508-3, Annex A and B, as accredited by the certification authority

verified with tolerable effort adhere to these structural properties anyway. In contrast, the modularity we do achieve is of a more *behavioural* nature: the effect of a function is summarised in its interface specification, even though the function body might be of arbitrary size and complexity.

Concerning Table A.9, we cover formal proofs and static analysis. The latter includes the measures marginal value analysis, control as well as data flow analysis and symbolic execution. Whereas our Hoare-logic style verification resembles a symbolic execution, many properties that are derived from the other analyses, like ensuring that only initialised variables are read, are also subsumed by formal verification.

However, most of the work in a verification effort goes into testing, so one would require that the overall amount of functional testing can be reduced in a development process using formal verification. This is confirmed in a remark in the standard (§ 7.4.7.2, Remark 2). In our case, the only tests that had to be performed on the module level were related to over-/underflow and numerical stability. No functional testing had to be performed for the formally verified units, due to the level of detail at which both specifications and the programming language are modelled.

7.5.4 Traceability

The V-model of IEC 61508-3 (cf. Fig. 2.4) asks for traceability between adjacent phases on the downward leg, i. e. from the system safety requirements down to the code, as well as ‘horizontal’ verification on the upward leg from code to the integrated and validated system, where appropriate tests ensure the satisfaction of all requirements. The model somewhat neglects model-based analysis and does not assign it a specific level; it might be considered part of the software architecture, but in any case has a direct link to the safety requirements. A definite strength of our methodology is the very strong link between this analysis level and the concrete source code (at the bottom of the V-model): Formal code verification in our methodology ensures both traceability between code and module design, *and* between module design and the analysis level. The main reason for this is the high level of abstraction of code specifications, in which the domain formalisation is directly embedded. For example, take the specification in Fig. 7.14 of the function computing a polygonal approximation of the curve described by a single point of the vehicle’s contour during an emergency stop. Its specification directly expresses that the area described by the returned polygon completely contains the braking curve in the two-dimensional environment model.

Chapter 8

Conclusion

8.1 Summary

In this thesis we have shown how formal functional specification and verification can be applied to enable the use of mathematically-oriented software in safety-related systems. We have presented the design and realisation of a formal verification environment that allows to establish a formal connection between high-level safety requirements and concrete source code. We have further demonstrated its application in the certification of a software module that is part of a safety-related system ensuring collision avoidance for automated guided vehicles.

We started by giving a brief overview of applicable law, in particular the EC machinery directive (Sec. 2.1.1), and pertinent safety standards, where we focused on IEC 61508, a standard applicable for the functional safety of programmable electronic devices. We highlighted the important concepts codified in the standard, especially the safety integrity level (SIL) which is used to classify the safety requirements to be considered for a concrete system (Sec. 2.1.2).

A functional specification language (CSI) was devised which is based on classical preconditions and postconditions of program functions. Specifications are attached to C source code through annotations. A unique feature of the language is the high level of abstraction and the conciseness of specifications which it affords. This is achieved by formalising the domain model derived from high-level safety requirements as Isabelle/HOL theories and by expressing source code specifications in terms of this model (Sec. 3.3.2). The connection between domain objects and program objects is established through the concept of representation functions. These are logic functions which interpret specific memory areas as the representations of domain objects. We furthermore devised binary combinators by which memory layouts, specifically aliasing patterns, can be described. These descriptions avoid combinatorial problems that are inherent in pairwise inequality constraints (Sec. 3.4).

The memory model for C is the crucial link between program values and logic values and determines the ease with which pointer programs can be verified. Our memory model is based on finite typed maps, i. e., functions that map locations to typed sequences of atomic values (Sec. 4.2). The included type information allowed us to neatly define the validity of pointer and array accesses (Sec. 4.2.6). The chosen representation format is largely compliant with the loose memory model of the C standard. Importantly however, our model allows for a restricted form of the split-heap property, where two valid locations denoting structure fields with different names are never aliased. The model further allows to treat local variables as first-class entities whose address can be taken, stored and passed to other functions, which was a key design goal. Rules for update simplification, i. e. the process of deciding whether and how to integrate a state update into a given memory-dependent logic value, were developed for both simple program values and representation functions, enabling a uniform and largely automatic treatment of all memory-dependent values during verification (Sec. 4.5).

The memory model and the formal denotational semantics for C programs (Chapter 5) provide the basis of the program logic by which programs are eventually verified to satisfy their specification. The structured specifications attached to C functions are encoded in simple Hoare-style assertions within the theorem prover. Program functions can be verified individually, relying only on the specifications of called functions, due to the definition of modular correctness (Sec. 6.2). The associated proof rules are of a rather administrative character and their application within the verification environment is fully automatic. The verification conditions thus generated are described by a simple grammar and represent the paths of a symbolic execution of the program (Sec. 6.4). This distinguishes our approach from related ones that also generate verification conditions, but where the connection to the actual source code is mostly lost. We finally described the special purpose tactics by which the theorem prover was extended. These are used by verifiers to simplify and eventually discharge the verification conditions (Sec. 6.5). The focus regarding automation is on memory update simplification, pointer validity and the deconstruction of the verification condition according to the grammar. Proofs of the domain-related parts have to be done mostly manually using Isabelle's regular proof support.

The verification environment was used in the SAMS project to satisfy several software verification objectives in an IEC 61508-3 certification. We presented the formalisation of high-level requirements in the domain model (Sec. 7.2) and showed concrete source code specifications of the algorithm computing safety zones (Sec. 7.3). Most functions of the safety-related software had been formally verified when they were handed to the certification authority. To assess our claim about the adequacy of formal verification for mathematically-oriented software, an evaluation was performed (Sec. 7.4) with respect to the errors found through formal verification, the influence it had on the development process as well as its technical and conceptual limitations.

Finally, we described how to integrate the verification environment into a safety-related software development process according to IEC 61508-3. We sketched the tool qualification and outlined the 'certification credit' obtained by using the tool. This specifically includes the reduction of functional testing on the module level and the coverage of several static analysis measures.

8.2 Concluding Remarks

Our general hypothesis was that the verification methodology described in this thesis contributes to the successful certification of computationally complex software and hence enables its use in safety-related systems. The results of the SAMS project support this claim. A certification authority confirmed that the safety-related software developed in the project, whose verification was largely performed according to this methodology, is ready for use in a system complying with the requirements of the safety standard IEC 61508-3 up to SIL 3. Moreover, it was acknowledged that the verification environment can be used to satisfy a specified number of verification objectives up to SIL 3.

Formal connection to high-level requirements. We identify two characteristics of the software we verified that are important when applying our methodology. First, the correct implementation of computationally complex behaviour is crucial for software safety. Contrast, for example, an algorithm that merely *chooses* a safety zone from a given small set of pre-defined, user-validated safety zones suitable for a given velocity with an algorithm that *computes* these zones. The latter's implementation is susceptible to vastly more errors and thus exacts a larger verification effort. The relevant properties go beyond the capabilities of automatic verification techniques and can only be verified by review, testing or deductive formal methods. Second, the safety requirements —such as the appropriateness of computed safety zones w. r. t. the braking model— are well-suited for a mathematically rigorous specification on the geometric level, which can be based on a corresponding domain formalisation. Many requirements about safety devices measuring distances and used for detection such as laser scanners, light curtains or ultrasonic sensors are of this kind. Formal functional verification still proves useful if requirements specified at a lower level are verified, although a shorter traceability link is established.

We note that the targeted safety requirements are essentially free from dynamic aspects. While the braking model arguably describes the dynamics of the EUC, these are determined in a simple manner by the *current state* of the EUC without any future interference by the safety component: once an emergency stop signal is raised, the EUC comes to a halt. We cannot handle safety requirements related to true dynamic aspects of the system. One example where dynamic aspects are at the heart of the safety requirements is given by Platzer and Clarke [128] who verify the behaviour of aircraft during collision avoidance manoeuvres.

In the robotics domain, where safety becomes an issue as robots and humans increasingly interact with each other, algorithms based on probabilistic approaches prevail [143]. This is because they crucially have to cope with uncertainty. In combination with the use of high-dimensional sensors like cameras even the formal specification of desired non-dynamic properties becomes hard. For example, we would not know how to specify that an algorithm reliably detects human faces on camera images.

Memory model. Our goal with respect to the formalisation of the memory model was rather pragmatic. Like other modern formalisations that are not based on separation logic [137, 145, 90, 56] we used the ideas of the Burstall-Bornat split heap memory model to exclude certain aliasing patterns, most importantly that between differently named structure fields, from the start. However, our memory model does not assign a separate memory area to each structure field, but stores structures en bloc. We are not sure if our initial assumption that the simplicity of the model would ease a tool qualification is actually valid. The certification authority made no comments in this direction. The probably more important fact is that the model remains largely faithful to the C standard. This simplifies a future integration of further memory aspects. The unique feature of our model —if compared to other approaches used in the verification of realistic programs— is that the use of local variables is not restricted. Emulating call-by-reference by passing pointers is a *very* common pattern in C programs, as witnessed by countless functions of the GNU C library[1]. It is also frequently used in the SAMS software. In contrast, modelling precision to the bit-level was not of great importance so far, as it certainly is in other domains like operating system verification as done in the Verisoft project.

Mathematical domain modelling in an interactive theorem prover. The tasks of visualisation, simulation and numerical computations are all essential parts of safety engineering and required to explore the problem space and to quickly validate approaches to a solution. Tools used for this purpose include computer algebra systems (CAS) like Maple or Mathematica and numerical computing environments like MATLAB.[2] These tools are also practically superior to theorem provers in the symbolic manipulation of expressions, in solving linear equation systems or differential equations, or in integrating and differentiating functions. However, the models yielded by these tools are semi-formal and less reliable than those developed in a theorem prover. CAS apply extremely complex and optimised algorithms which are known to contain bugs.[3] We therefore propose that initial models be developed and analysed semi-formally, and that the formalisation in the theorem prover be done when the design has stabilised. This ultimately leads to formally documented, repeatable, machine-checked proofs of the safety-related properties of the system to be developed.

A possibility for a deeper cooperation of unsafe but powerful tools and theorem provers lies in a 'guess and verify' approach, where the output of the former is verified by the latter. This is sensible in cases where verifying a result is drastically simpler than obtaining the result. Solutions of linear equation systems are an example. This idea has been successfully applied by Harrison [71], who uses numerical algorithms for semidefinite programming to obtain sums of squares representations for certain real polynomials and verifies their equivalence in a theorem prover.

Finally, the formalisation of a geometry-based model requires that appropriate definitions and theorems of real analysis are available. Standard theorems of real analysis such as Rolle's theorem, the mean value theorem, or even the Hahn-Banach theorem have all been formalised in Isabelle. However, most applications of Isabelle are concerned with discrete mathematics and the set of theorems about real analysis is fragmentary. We had to prove rather foundational theorems about elementary functions ourselves. We therefore advocate a top-down approach to domain modelling where missing theorems are merely

[1] `http://www.gnu.org/software/libc`

[2] `http://www.wolfram.com`, `http://www.maplesoft.com`, `http://www.mathworks.com`

[3] For example, a known bug of Mathematica 7.0 was that it evaluated $\sqrt{x^2} = x$ to *True*, while at the same time being able to find an instance for which $\sqrt{x^2} \neq x$ holds (cf. the Usenet discussion at `http://groups.google.com/group/comp.soft-sys.math.mathematica/msg/f54913012cd2e8f7?pli=1`).

assumed initially and proven later, when their necessity is definite. In our experience this reduced the number of 'orphan' theorems whose proof consumed time but which were not needed for the specification and verification of programs.

8.3 Future Work

The confirmation by a certification authority that the verification environment is compliant with IEC 61508-3 up to SIL 3 represents a satisfactory milestone in its development. However, several extensions and improvements are imaginable.

Qualification for other standards It would be interesting to apply our verification environment in a project in which software is certified against the arguably most demanding software safety standard DO-178B. The standard explicitly states that the software verification process objectives are satisfied by analyses, reviews, and the development and execution of test cases. The largest challenge would therefore be to obtain certification credit for requirements that must originally be satisfied by testing. A recent publication [31] expresses doubt that this will be possible.

Usability It would be very useful to attach more 'extra-logical' structure to the generated verification conditions. The discussed encoding via dummy predicates provides boundless opportunities such as the mapping of symbolic program states to corresponding line numbers in the source code. Another idea is to categorise assumptions ('technical', 'postcondition', 'domain-level property') and to display them accordingly. To this end, the matured user interface ProofGeneral should be dropped, possibly in exchange for Gast's new I3P interface [63], which emphasises extensibility and has a modern look and feel.

Integration of automatic tools The degree of automation that is achieved w. r. t. update simplification and, more importantly, regarding arithmetic proof goals needs to be improved. Moreover, Isabelle's reasoning tools like the simplifier become slow in the face of hundreds of assumptions. This is a use case where SMT solvers shine. However, their use is in slight opposition to the high abstraction level of specifications that we promote, since the theories they support are rather weak. For example, most of them do not even support simple set algebra. The approach of FRAMA-C and of Zee et al. [153] is (very) roughly analogous to repeatedly applying the `pr_step` tactic and then passing the resulting verification conditions to a whole collection of provers, with the hope that each VC will be verified by at least one prover. Isabelle provides the infrastructure (named Sledgehammer) to implement this solution by allowing to delegate sub-proofs to other provers.

Deriving test cases from specifications Writing formal specifications is a labour intensive task. Its value would increase if specifications were re-used to automatically generate test cases and to obtain an 'oracle' determining whether a test case passed or failed. Universal quantifiers naturally lend themselves to being used as generators: for testing, $\forall x.\ P(x) \longrightarrow Q(x)$ can be seen as the operation of generating items x that satisfy P (via a

P-generator) and testing whether all of them satisfy *Q*. This is an idea that is also exploited in the QuickCheck tool [38, 39].

Hybrid systems An early idea [61] similar to the modelling of hybrid systems, which has not been followed subsequently in the SAMS project, was to use the semantics of program functions as formal objects whose properties are described by their verified specification. These objects would describe a discrete system which reacts to its inputs in specified time intervals by yielding new outputs. The domain model would be lifted by one dimension by turning all entities into time-dependent functions. The behaviour of some of these entities, such as the motion of the EUC, would be influenced by the output of the discrete system. By proving properties about the behaviour of the robot controlled by the software in this model we would eventually be able to establish a formal connection between the source code and time-related high-level requirements.

Appendix A

Isabelle/HOL Theory Graph

The Isabelle/HOL theories developed in the SAMS project are comprised of the theories implementing the verification environment and those concerned with the domain modelling. Fig. A.1 depicts all involved theories. It can be seen that the two strands are entirely separated until joined in theory *RecordRepr*, which defines representation functions. Their dependency on both strands stems from the fact that they define mappings from memory areas to interpretations in the domain. The following table briefly summarises the contents of the theories, except for *Polynomial* and *Poly-Deriv* which are pre-defined by Isabelle/HOL and are concerned with univariate polynomials and their differentiation.

Theory name	Description
Auxiliaries	Generally helpful theorems and definitions without specific relation to the verification environment
State	The memory model
ILC	Abstract syntax (datatypes) for C programs
Selection	Definitions and theorems related to array access and field selection
Parameters	Correspondence between function parameter types and argument values
Modifies	Non-atomic state modification (**@modifies** clauses)
ST	State transformers as monads (type *'a ST*) and basic monadic operations
Spec	State predicates and relations (types *'a SP* and *'a SR*) and their satisfaction relation w. r. t. state transformers
Env	Environments Γ for program variables, function specifications and function semantics
SpecILC	Instantiation of the datatypes of *ILC* to concrete specification types
Sem	Semantics of lvalues, expressions, statements, functions, and specification items

SemLemmas	Equivalence of full and side-effect free semantics under certain conditions
ModularCorrectness	Correctness of C programs w. r. t. their specifications and the relation to environment extensions
ProofRulesSetup	Definition of Hoare triples for lvalues, expressions, etc., and preparatory lemmas for correctness proof of proof rule for **while** statement
ProofRules	93 theorems comprising the set of proof rules used to derive the initial verification condition
TacticLemmas	Auxiliary theorems required by automatic proof tactics
SAMS	Theory setting up the proof tactics
SAMSTools	Auxiliary theorems related to differentiation
SAMSMonoton	Monotone functions and their derivations
SAMSKonvex	Convex sets of points
Trigonometrie	Theorems over trigonometric functions
SAMSDomain	Definitions and theorems of domain-related concepts (rigid body transformations, braking behaviour, braking measurements, circular arcs)
SAMSAlphalinien	Theorems about the relation between velocity space (v, ω) and braking configuration space (s, α)
SAMSPufferradius	idem
Bogenhuelle	Theorems about arc approximation by polygonal lines
RecordRepr	Representation functions for the aggregate C types used in the SAMS project

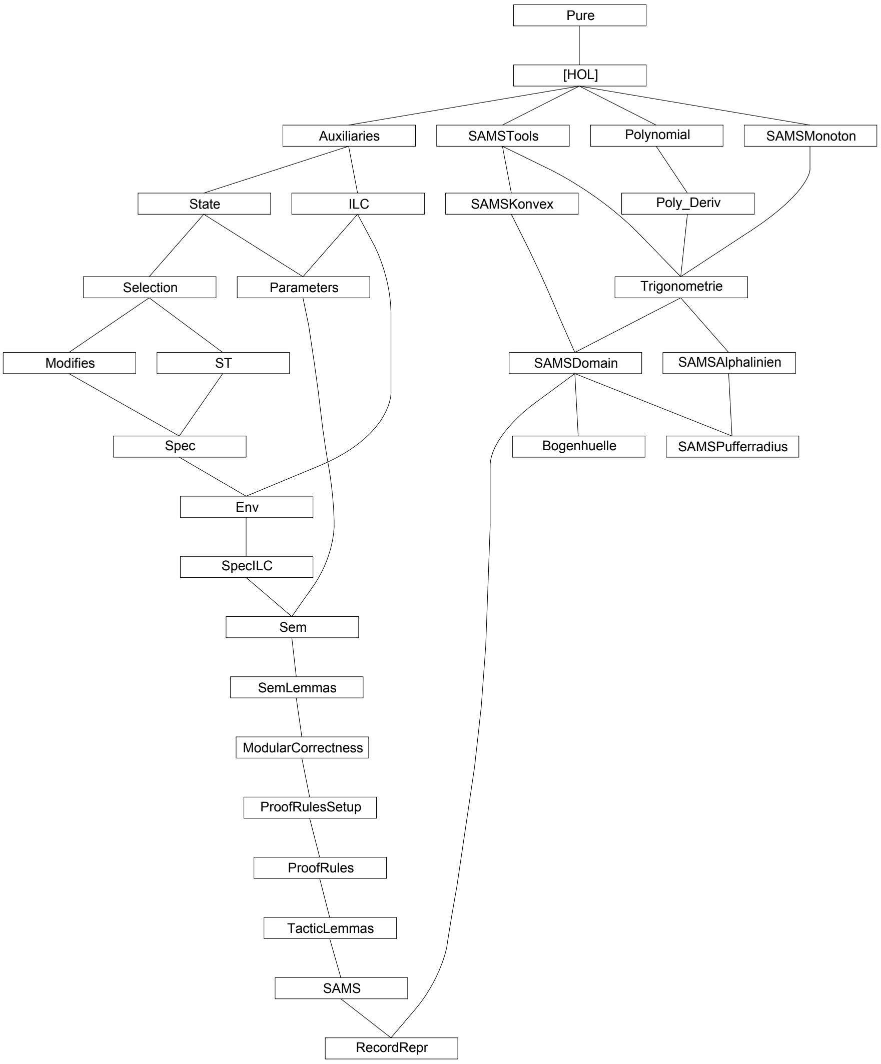


Figure A.1: Isabelle/HOL theories of the verification environment (left strand) and the SAMS domain (right strand)

Appendix B

Concrete Code Examples

B.1 Example: Initial Verification Condition

In this section we present the full proof state introduced by the execution of the tactic *prtac* deriving the initial verification condition for a given representation of a C function, as promised in Sec. 6.4.1.

The function add_rotate, shown below, is supposed to perform the following mathematical operation:

$$q = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} (r + p') \qquad \text{where } p' = \begin{cases} 0 & \text{if } p = \text{NULL} \\ p & \text{otherwise} \end{cases} \tag{B.1}$$

The result vector q shall equal the vector obtained by rotating the sum of r and p by 90 degrees. If the pointer for p is **NULL**, only r shall be rotated.

In the following, we present the complete contents of a file `vec.c` in which this operation is implemented. The initial type definition and file-scope annotations introduce the type of vectors, make the relevant theories (*RecordRepr* and *Rotation*) visible in the theory generated by the front-end, and bind the representation function *Vec2DR* (cf. Sec. 4.5.1 for its definition) to the type vec2d of vector representations.

```
typedef struct Vec2DStruct {
    double x; double y;
} vec2D;

/*@ @theory RecordRepr @*/
/*@ @theory Rotation @*/
/*@ ^function _Any Vec2DR(vec2D *p); @*/
```

The remaining content of `vec.c` comprises the specification and definition of function add_rotate. The functions expects three pointers p, r, q, pointing to vectors denoted by the same name in Eq. (B.1). The function specification directly corresponds to Eq. (B.1), with the necessary additional constraints on pointer validity. *rot-left* performs the rotation by 90 degrees, and vector addition is denoted by the overloaded infix + operator. The implementation of the function is rather obvious and requires no explanation. We have added a **@join** annotation for the **if** statement that ensures non-nullity of p to avoid a duplication of the postcondition in the initial verification condition.

```
/*@
  @requires (\valid(p) || p == NULL) &&
    \valid(r) && \valid(q) && r != q && p != q
  @modifies *q
  @ensures ${ ^Vec2DR{q} = rot_left (^Vec2DR{r} +
              (if `{p == NULL} then 0 else ^Vec2DR{p})) }
  @*/
void add_rotate(vec2D *p, vec2D *r, vec2D *q) {
    vec2D z = {0.0, 0.0};
    /*@
      @join (\if \old(p) == 0
             \then p == &z
             \else p == \old(p)) && \valid(p)
      @modifies p
      @*/
    if (p == 0) { p = &z; }

    q->x = -(p->y + r->y);
    q->y = p->x + r->x;
    return;
}
```

To demonstrate that add_rotate satisfies its specification (cf. Sec. 6.2.1) the following lemma must be proven

```
lemma
  "correct_function_modular sams_genv add_rotate_def"
```

where *sams-genv* is an environment mapping only the identifier *add-rotate* to the specification of this function, and being undefined everywhere else. By applying the proof tactic *prtac* to this proof goal, the initial verification condition is computed. It is shown in the framed box below, and has been output directly by Isabelle. Therefore, the proof state looks exactly as it does when a verifier views it inside the user interface of the theorem prover. The notation is consistent with that used in this thesis, except for a few shorthands; for example, the formally incorrect $\Uparrow_{"x"}$ is displayed instead of the longer $\Uparrow_{\nu\cdot"x"_n}$, which correctly indicates the dependence on the variable index n.

The initial state S_1 and several intermediate states arising from the initialisation of the function parameters are bound on the meta-level already (S, S', $S'a$ to $S'd$). The relation between these states is maintained by according assumptions, e. g., $S_1 \otimes (\nu \cdot "p"_n, \mathit{PtrVal}\ p, \mathit{Vec2D\text{-}t}\ *) \doteq S'$. Furthermore, the facts that all local variables and parameters are fresh for the initial state S_1 has been derived. Finally, the validity requirements of the precondition have been translated to $\bullet(p_{\mathit{Vec2D\text{-}t}} |\ S_1)$ etc., and the pointer values (locations) p and r are assumed to be inequal to q.

The conclusion of the proof goal is a term over the grammar of Eq. (6.4).

```
(⋀Γ S1 p r q S n S' na S'a nb S'b nc S'c S'd.
   ⟦sams_genv ◁ Γ; ''p''^ν ∉_S S1; S1⊗(''p''^ν, PtrVal p, Vec2D_t *) ≐ S';
    ''r''^ν ∉_S S'; S'⊗(''r''^ν, PtrVal r, Vec2D_t *) ≐ S'a; ''q''^ν ∉_S S'a;
    S'a⊗(''q''^ν, PtrVal q, Vec2D_t *) ≐ S'b; ''z''^ν ∉_S S'b;
    S'b⊕(''z''^ν, Vec2D_t) ≐ S'c;
    S'c(''z''^ν ::= PtrVal (''z''^ν_Vec2D_t→''x'')) ≐ S'd; S = S1;
    •(p_Vec2D_t|S1) ∨ p = NULL; •(r_Vec2D_t|S1); •(q_Vec2D_t|S1); r ≠ q;
    p ≠ q; ''z''^ν ∉_S S1; ''q''^ν ∉_S S1; ''r''^ν ∉_S S1; True⟧
   ⟹ (∀S'. S'd(''z''^ν_Vec2D_t→''x'' ::= DoubleVal 0) = S' ⟶
            (∀S'a. S'(''z''^ν_Vec2D_t→''y'' ::= DoubleVal 0) = S'a ⟶
                  (∀Λ'. {q_Vec2D_t→''y'', q_Vec2D_t→''x''} ∪ ⇑''p'' ∪
                       ⇑''r'' ∪
                       ⇑''q'' ∪
                       ⇑''z'' =
                       Λ' ⟶
                       (∀M. ⇑''p'' = M ⟶
                            (if ''p''^ν @l S'a = NULL
                             then ∀S'.
 S'a(''p''^ν ::= PtrVal ''z''^ν) = S' ⟶
 (if ''p''^ν @l S'a = NULL then ''p''^ν @l S' = ''z''^ν
  else ''p''^ν @l S' = ''p''^ν @l S'a) ∧
 •((''p''^ν @l S')_Vec2D_t|S')
                             else •((''p''^ν @l S'a)_Vec2D_t|S'a)) ∧
                            M ⊆ Λ' ∧
                            (∀T. S'a ⊑_M T ⟶
                                 (if ''p''^ν @l S'a = NULL
                                  then ''p''^ν @l T = ''z''^ν
                                  else ''p''^ν @l T = ''p''^ν @l S'a) ∧
                                 •((''p''^ν @l T)_Vec2D_t|T) ⟶
                                 (let rv = ''q''^ν @l T
                                  in ((let rva = ''p''^ν @l T
   in (let dv = rva_Vec2D_t→''y'' @d T; rva = ''r''^ν @l T
      in (let dva = rva_Vec2D_t→''y'' @d T
         in ∀S'. T(rv_Vec2D_t→''x'' ::= DoubleVal (- (dv + dva))) = S' ⟶
                (let rv = ''q''^ν @l S'
                 in ((let rva = ''p''^ν @l S'
                     in (let dv = rva_Vec2D_t→''x'' @d S';
                               rva = ''r''^ν @l S'
                        in (let dva = rva_Vec2D_t→''x'' @d S'
                           in ∀S'a.
 S'(rv_Vec2D_t→''y'' ::= DoubleVal (dv + dva)) = S'a ⟶
 (∀S'. S'a⊖''z''^ν = S' ⟶
       (∀S'a. S'⊖''q''^ν = S'a ⟶
             (∀S'. S'a⊖''r''^ν = S' ⟶
                  (∀S'a. S'⊖''p''^ν = S'a ⟶
                        (if p = NULL
                         then Vec2DR S'a q = rot_left (Vec2DR S'a r)
                         else Vec2DR S'a q =
                              rot_left
                               (Vec2DR S'a r + Vec2DR S'a p))))))) ∧
•(rva Vec2D_t|S')) ∧
                                               •(rva Vec2D_t|S')) ∧
                                        rv Vec2D_t→''y'' ∈ Λ') ∧
                                      •(rv Vec2D_t|S'))) ∧
                  •(rva Vec2D_t|T)) ∧
           •(rva Vec2D_t|T)) ∧
    rv Vec2D_t→''x'' ∈ Λ') ∧
   •(rv Vec2D_t|T)))))) ∧
                        ''z''ν Vec2D_t→''y''
                        ∈ {q Vec2D_t→''y'', q Vec2D_t→''x''} ∪ ⇑''p'' ∪ ⇑''r'' ∪ ⇑''q'' ∪
                            ⇑''z'') ∧
             ''z''ν Vec2D_t→''x''
             ∈ {q Vec2D_t→''y'', q Vec2D_t→''x''} ∪ ⇑''p'' ∪ ⇑''r'' ∪ ⇑''q'' ∪
                ⇑''z'')
```

B.2 Implementation of Braking Configuration Functions

The code snippets in this and the next section are taken literally from the SAMS project code. (Only the indentation has been adjusted to fit the code onto the page.) They are shown here for reference purposes.

The following function implements the binary search routine that finds the index at which the upper bound of the interval resides in which the input velocity v is contained in the array of braking measurements sams_konfiguration.bremswege.messungen.

```
Int32 bin_suche_index_v( Float32 v )
{
  Int32 i;
  Int32 imax = 0;
  Int32 imin = sams_konfiguration.bremswege.anzahl-1;

  /*@
    @invariant 0 <= imax && imax < imin &&
        imin < sams_konfiguration.bremswege.anzahl &&
        sams_konfiguration.bremswege.messungen[imax].v > v &&
        v >= sams_konfiguration.bremswege.messungen[imin].v
    @modifies i, imin, imax
    @variant imin - imax
    @*/
  while (imin-imax > 1) {
    i = (imax+imin) / 2;
    if (v >= sams_konfiguration.bremswege.messungen[i].v) {
      imin = i;
    }
    else {
      imax = i;
    }
  }
  return imin;
}
```

The following function yields the quotient of the braking distance covered during a braking in forward motion starting from velocity v, and v itself.

```
Float32 bremsweg_geradeaus( Float32 v )
{
  Float32 ret;
  Int32 iv;
  const Bremsmessung * const m =
    sams_konfiguration.bremswege.messungen;

  /*@
    @join v == $fabs(\old v)
    @modifies v
    @*/
  if (v < 0.0) {
    v = -v;
  }

  if (v >= m[0].v) {
    ret = m[0].s / (m[0].v*m[0].v*m[0].v) * v*v;
  }
  else
  {
    iv = bin_suche_index_v(v);
    if (iv == sams_konfiguration.bremswege.anzahl-1) {
        ret = m[iv-1].s / m[iv-1].v;
    }
    else {
      ret = ((m[iv-1].s - m[iv].s) / (m[iv-1].v - m[iv].v) *
              (v - m[iv].v) + m[iv].s) / v;
    }
  }
  return ret;
}
```

B.3 Implementation of the Arc Hull Function

The following is the definition of the function bogenhuelle_L computing the approximation of an arc using L auxiliary points. Its verification has been discussed in Sec. 7.3.2.

```
SAMSStatus bogenhuelle_L( Laenge s, WinkelRad alpha, Int32 l,
     const Vektor2D *startpunkte_daten, Int32 startpunkte_laenge,
     Vektor2D *ergebnis_daten, Int32 ergebnis_laenge_max )
{
  StarrkoerperTransformation bogen_k;
  Int32              k;
  Laenge             s_1, s_k;
  WinkelRad          alpha_1, alpha_k;
  Vektor2D           endpunkte_b1[SAMS_ROBOTERKONTUR__ARRSZ];

  SAMSStatus         ret = sams_sicher;
  /* Execution flow logging */
  pak_funktionseintritt( pakn_bogenhuelle_L );

  if (l < 1)
  {
      ret = sams_systemfehler;
      schreibe_ereignis_int( err_huelleL_min1pkt, l );
  }
  else if ((startpunkte_laenge > SAMS_ROBOTERKONTUR__ARRSZ) ||
           (l * startpunkte_laenge > ergebnis_laenge_max))
  {
      ret = sams_systemfehler;
      schreibe_ereignis( err_arraygrenze_ueberschritten );
  }
  else
  {
    /* All arcs have length s/l and angle alpha/l. */
    s_1 = s / (Float32)l;
    alpha_1 = alpha / (Float32)l;
    /* Compute end points. */
    bremskonfiguration_zu_skt( s_1, alpha_1, &bogen_k );
    ret = transformiere( &bogen_k,
                         startpunkte_daten,
                         startpunkte_laenge,
                         endpunkte_b1,
                         SAMS_ROBOTERKONTUR__ARRSZ );
    if (ret == sams_sicher)
    {
      /* Compute additional points of first arc. */
      ret = bogenhuelle_1( alpha_1,
                           startpunkte_daten,
                           endpunkte_b1,
                           startpunkte_laenge,
                           ergebnis_daten,
                           ergebnis_laenge_max );
after_b1:
      /* Compute further arcs by transforming first. */
      /*@
        @invariant 1 <= k && k <= l &&
          (ret == sams_sicher -->
            @after_b1(ret) == sams_sicher) &&
          ${ ∀ (i :: DomInt). 0 ≤ i ∧ i < ‘startpunkte_laenge ⟶
             ( ∀ (j :: DomInt). 0 ≤ j ∧ j < ‘k ⟶
               transformiere
                 (bogentransformation
                      (‘s * real j / real ‘l)
                      (‘alpha * real j / real ‘l))
                   ^Vektor2DR{ergebnis_daten[$i]} =
               ^Vektor2DR{ergebnis_daten[$i + $j *
                           startpunkte_laenge]})
            }
        @modifies sams_andere, ret, k, s_k, alpha_k, bogen_k,
          ergebnis_daten[startpunkte_laenge
                         : startpunkte_laenge * l]
        @variant l - k
        @*/
      for ( k = 1; (ret == sams_sicher) && (k < l); ++k )
      {
        s_k = (s * (Float32)k) / (Float32)l;
        alpha_k = (alpha * (Float32)k) / (Float32)l;
        /* Transformation arc 1 to arc k */
        bremskonfiguration_zu_skt( s_k, alpha_k, &bogen_k );
        ret = transformiere( &bogen_k,
                             ergebnis_daten,
                             startpunkte_laenge,
                             &ergebnis_daten[k *
                                startpunkte_laenge],
                             ergebnis_laenge_max - k *
                                startpunkte_laenge );
      }
    }
  }

  /* Execution flow logging */
  pak_funktionsaustritt( pakn_bogenhuelle_L );
  return ret;
}
```

Bibliography

[1] M. Abadi, L. Cardelli, and P. L. Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121(1-2):9–58, 1993. doi: 10.1016/0304-3975(93)90082-5.

[2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. doi: 10.1016/0304-3975(91)90224-P.

[3] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005. doi: 10.1007/s10270-004-0058-x.

[4] Eyad Alkassar and Mark A. Hillebrand. Formal functional verification of device drivers. In Jim Woodcock and Natarajan Shankar, editors, *Verified Software: Theories, Tools, Experiments Second International Conference, VSTTE 2008*, volume 5295 of *Lecture Notes in Computer Science*, pages 225–239, Toronto, Canada, October 2008. Springer. doi: 10.1007/978-3-540-87873-5_19.

[5] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. The verisoft approach to systems verification. In Natarajan Shankar and Jim Woodcock, editors, *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*, volume 5295 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2008. doi: 10.1007/978-3-540-87873-5_18.

[6] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995. doi: 10.1016/0304-3975(94)00202-T.

[7] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. doi: 10.1016/0304-3975(94)90010-8.

[8] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resources. *Theoretical Computer Science*, 389(3):411–445, 2007. doi: 10.1016/j.tcs.2007.09.003.

[9] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer.

[10] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 363–366. Springer, 2000. doi: 10.1007/3-540-46428-X_25.

[11] Jarnet Barnes, Rod Chapman, Randy Johnson, James Widmaier, David Cooper, and Bill Everett. Engineering the tokeneer enclave protection software. In *Intl. Symp. on Secure Software Engineering (ISSSE'06)*. IEEE Computer Society, 2006.

[12] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003.

[13] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2005. doi: 10.1007/b105030.

[14] Mike Barnett, Bor-Yuh Chang, Robert DeLine, Bart Jacobs, and K. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2006. doi: 10.1007/11804192_17.

[15] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, July 2007. Berlin, Germany.

[16] Nurlida Basir, Ewen Denney, and Bernd Fischer. Constructing a safety case for automatically generated code from formal program verification information. In *27th Int. Conference on Computer Safety, Reliability, and Security (SAFECOMP'08)*, volume 5219 of *Lecture Notes in Computer Science*, pages 249–262. Springer, 2008. doi: 10.1007/978-3-540-87698-4.

[17] Nurlida Basir, Ewen Denney, and Bernd Fischer. Deriving safety cases for hierarchical structure in model-based development. In Erwin Schoitsch, editor, *29th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP 2010)*, volume 6351 of *Lecture Notes in Computer Science*, pages 68–81. Springer, 2010. doi: 10.1007/978-3-642-15651-9_6.

[18] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI C specification language. `http://frama-c.cea.fr/download/acsl_1.4.pdf`, October 2008. Preliminary design, version 1.4.

[19] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.

[20] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–312, London, UK, 2001. Springer.

[21] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991. doi: 10.1109/32.75415.

[22] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010. doi: 10.1145/1646353.1646374.

[23] Michel Bidoit and Peter D. Mosses. *CASL User Manual*, volume 2900 of *Lecture Notes in Computer Science*. Springer, 2004.

[24] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL'06*, pages 55–66. ACM, 2006. doi: 10.1145/1111037.1111043.

[25] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[26] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. doi: 10.1007/s10817-009-9148-3.

[27] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987. doi: 10.1016/0169-7552(87)90085-7.

[28] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *Software Engineering, IEEE Transactions on*, 21(10):785–798, Oct 1995. doi: 10.1109/32.469460.

[29] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[30] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer, 2001. doi: 10.1007/3-540-45510-8_9.

[31] Duncan Brown, Hervé Delseny, Kelly Hayhurst, and Virginie Wiels. Guidance for using formal methods in a certification context. In *Embedded Real Time Software and Systems*, Toulouse, France, May 2010.

[32] Manfred Broy and Andreas Rausch. Das neue V-Modell XT. *Informatik-Spektrum*, 28(3):220–229, 2005. doi: 10.1007/s00287-005-0488-z.

[33] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005. doi: 10.1007/s10009-004-0167-4.

[34] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In Bernard Meltzer and Donald Michie, editors, *Proceedings of the Seventh Annual Machine Intelligence Workshop*, volume 7 of *Machine Intelligence*, pages 23–50. Edinburgh University Press, 1972.

[35] Robert Cartwright and Derek Oppen. The logic of aliasing. *Acta Informatica*, 15(4):365–384, August 1981. doi: 10.1007/BF00264535.

[36] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, 25:129–166, 2004. ISSN 0925-9856. doi: 10.1023/B:FORM.0000040026.56959.91.

[37] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978. doi: 10.1109/TSE.1978.231496.

[38] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM. doi: 10.1145/351240.351266.

[39] Koen Claessen and John Hughes. Testing monadic code with quickcheck. *SIGPLAN Notices*, 37(12):47–59, 2002. doi: 10.1145/636517.636527.

[40] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50:752–794, September 2003. ISSN 0004-5411. doi: 10.1145/876638.876643.

[41] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. doi: 10.1007/b96393.

[42] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[43] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, Munich, Germany, 2009. Springer.

[44] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A precise yet efficient memory model for C. In *4th International Workshop on Systems Software Verification (SSV 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier Science B.V., 2009.

[45] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 2010. doi: 10.1007/978-3-642-14295-6_42.

[46] Mirko Conrad. Testing-based translation validation of generated code in the context of IEC 61508. *Formal Methods in System Design*, 35:389–401, 2009. doi: 10.1007/s10703-009-0082-0.

[47] Mirko Conrad, Patrick Munier, and Frank Rauch. Qualifying software tools according to ISO 26262. In *Modellbasierte Entwicklung eingebetteter Systeme VI (MBEES)*, pages 117–128. fortiss GmbH, München, 2010.

[48] P. Cousot. Proving the absence of run-time errors in safety-critical avionics code. In C. Kirsch and R. Wilhelm, editors, *Proceedings of the Seventh ACM & IEEE International Conference on Embedded Sofware, Embedded Systems Week, (EMSOFT 2007)*, pages 7–9, Salzburg, Austria, September 2007. ACM press.

[49] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512973.

[50] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, New York, NY, USA, 1989. ACM. doi: 10.1145/75277.75280.

[51] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[52] DIN. EN ISO 13849-1:2006: Sicherheit von Maschinen – Sicherheitsbezogene Teile von Steuerungen – Teil 1: Allgemeine Gestaltungsleitsätze, 2006. Deutsches Institut für Normung e.V., Berlin.

[53] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of*

Integrated Circuits and Systems, IEEE Transactions on, 27(7):1165–1178, July 2008. doi: 10.1109/TCAD.2008.923410.

[54] The European Parliament and the Council. Directive 2006/42/EC. *Official Journal of the European Union*, L 157, 9 June 2006.

[55] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[56] Jean-Christophe Filliâtre and Claude Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, November 2004. Springer.

[57] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 193–205, New York, NY, USA, 2001. ACM Press. doi: 10.1145/360204.360220.

[58] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press. doi: 10.1145/512529.512558.

[59] Robert Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, pages 19–32, 1967.

[60] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics and Automation Magazine*, 4(1):23–33, 1997.

[61] Udo Frese, Daniel Hausmann, Christoph Lüth, Holger Täubig, and Dennis Walter. The importance of being formal. In Hardi Hungar, editor, *Proc. SafeCert 2008*, volume 238 of *Electronic Notes in Theoretical Computer Science*, pages 57–70. Elsevier Science, 2008. doi: 10.1016/j.entcs.2009.09.006.

[62] Holger Gast. Reasoning about memory layouts. In *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 628–643. Springer, 2009. doi: 10.1007/978-3-642-05089-3.

[63] Holger Gast. Towards a modular extensible Isabelle interface. In *TPHOLs'09*, 2009. Emerging Trends Section.

[64] Tilmann Glötzner. IEC 61508 certification of a code generator. In *System Safety, 3rd IET International Conference on*, pages 1–4, 2008.

[65] Joseph Goguen and Grant Malcolm, editors. *Software engineering with OBJ : algebraic specification in action.* Kluwer Academic Publishers, 2000.

[66] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.

[67] Mike Gordon. From LCF to HOL: a short history. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 169–185. MIT Press, Cambridge, MA, USA, 2000.

[68] Yuri Gurevich and James K. Huggins. The semantics of the C programming language. In *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*, pages 274–308. Springer, 1993. doi: 10.1007/3-540-56992-8.

[69] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[70] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. doi: 10.1016/0167-6423(87)90035-9.

[71] John Harrison. Verifying nonlinear real formulas via sums of squares. In Klaus Schneider and Jens Brandt, editors, *Proc. of the 20th Int. Conference on Theorem Proving in Higher Order Logics, TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 102–118. Springer, 2007.

[72] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-01, School of Electrical Engineering and Computer Science, University of Central Florida, 2009. Submitted for publication.

[73] Les Hatton. *Safer C*. Int. Series in Software Engineering. McGraw-Hill Book Company, 1995.

[74] Constance Heitmeyer, Myla Archer, Elizabeth Leonard, and John McLean. Applying formal methods to a certifiably secure software system. *IEEE Transactions on Software Engineering*, 34:82–98, 2008. doi: 10.1109/TSE.2007.70772.

[75] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with Blast. In *Proc. of the Tenth Int. Workshop on Model Checking of Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239, 2003. doi: 10.1007/3-540-44829-2_17.

[76] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259.

[77] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall Int. Ltd., 1985.

[78] C.A.R. Hoare. Viewpoint retrospective: an axiomatic basis for computer programming. *Communications of the ACM*, 52(10):30–32, 2009. doi: 10.1145/1562764.1562779.

[79] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303. Springer, 2000. doi: 10.1007/3-540-46428-X_20.

[80] IEC. *IEC 61508 – Functional safety of electrical/electronic/programmable electronic safety-related systems.* International Electrotechnical Commission, Geneva, Switzerland, 2000.

[81] ISO/IEC. *ISO/IEC 9899:1990 – Programming languages – C.* International Organization for Standardization/International Electrotechnical Commission, Geneva, Switzerland, 1990.

[82] ISO/IEC. *ISO/IEC 9899:1999 – Programming languages – C.* International Organization for Standardization/International Electrotechnical Commission, Geneva, Switzerland, 1999.

[83] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002. doi: 10.1145/505145.505149.

[84] Bart Jacobs and Erik Poll. Coalgebras and monads in the semantics of Java. *Theoretical Computer Science*, 291(3):329–349, 2003. doi: 10.1016/S0304-3975(02)00366-3.

[85] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):1–54, 2009. doi: 10.1145/1592434.1592438.

[86] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 2nd edition, 1990.

[87] Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. Formal verification of a flash memory device driver – an experience report. In *Model Checking Software*, volume 5156 of *Lecture Notes in Computer Science*, pages 144–159. Springer, 2008. doi: 10.1007/978-3-540-85114-1_12.

[88] Gerwin Klein. Operating system verification – an overview. *SADHANA – Academy Proceedings in Engineering Sciences*, 34:27–70, Feb 2009.

[89] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[90] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, Jun 2010.

[91] Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, December 1999. doi: 10.1007/s001650050057.

[92] Andrew Kornecki and Janusz Zalewski. Certification of software for real-time safety-critical systems: state of the art. *Innovations in Systems and Software Engineering*, 5:149–161, 2009. ISSN 1614-5046. doi: 10.1007/s11334-009-0088-1.

[93] A. Lankenau and T. Röfer. A safe and versatile mobility assistant. *Reinventing the Wheelchair. IEEE Robotics and Automation Magazine*, 8(1): 29–37, 2001.

[94] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006. doi: 10.1145/1127878.1127884.

[95] Dirk Leinenbach and Elena Petrova. Pervasive compiler verification - from verified programs to verified systems. In *Proc. 3rd Int. Workshop on Systems Software Verification (SSV 2008)*, volume 217 of *Electronic Notes in Theoretical Computer Science*, pages 23–40, 2008. doi: 10.1016/j.entcs.2008.06.040.

[96] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[97] Helge Löding and Jan Peleska. Symbolic and abstract interpretation for C/C++ programs. In *Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008)*, volume 217 of *Electronic Notes in Theoretical Computer Science*, pages 113–131, 2008. doi: 10.1016/j.entcs.2008.06.045.

[98] Alexey Loginov, Thomas Reps, and Mooly Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *Static Analysis Symposium*, volume 4134 of *Lecture Notes in Computer Science*, pages 261–279. Springer, 2006. doi: 10.1007/11823230_17.

[99] Damte Ltd. Guide to the new Machinery Directive 2006/42/EC. http://www.machinebuilding.net/, 2009. Retrieved Nov 2009.

[100] Christoph Lüth and Dennis Walter. Certifiable specification and verification of C programs. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009, Proceedings*, volume 5850, pages 419–434. Springer, 2009. doi: 10.1007/978-3-642-05089-3_27.

[101] C. Lüth and B. Wolff. TAS – a generic window inference system. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, number 1869 in Lecture Notes in Computer Science, pages 405–422. Springer, 2000.

[102] C. Marché, C. Paulin-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. doi: 10.1016/j.jlap.2003.07.006.

[103] Hennes Märtins. Statische Analyse von C-Programmen auf Einhaltung der MISRA-C-Richtlinien. Master's thesis, Universität Bremen, May 2010.

[104] John McHale. Upgrade to DO-178B certification – DO-178C – to address modern avionics software trends. Avionics Intelligence, `http://avi.pennnet.com`, Oct 2009. Retrieved Nov 2009.

[105] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.

[106] B. Meyer. Applying design by contract. *Computer*, 25(10):40–51, Oct 1992. doi: 10.1109/2.161279.

[107] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[108] J. Minguez and L. Montano. Nearness diagram (ND) navigation: Collision avoidance in troublesome scenarios. *IEEE Transactions on Robotics and Automation*, 20(1):45–59, 2004.

[109] MISRA. *MISRA-C:2004 – Guidelines for the use of the C language in critical systems*. Motor Industry Research Association (MIRA) Limited, Nuneaton, UK, 2004.

[110] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. doi: 10.1016/0890-5401(91)90052-4.

[111] O. Müller and K. Slind. Treating Partiality in a Logic of Total Functions. *The Computer Journal*, 40(10):640–651, 1997. doi: 10.1093/comjnl/40.10.640.

[112] Glenford J. Myers, Tom Badgett, Todd M. Thomas, and Corey Sandler. *The Art of Software Testing*. John Wiley and Sons, 2004.

[113] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, pages 209–265. Springer, 2002. doi: 10.1007/3-540-45937-5_16.

[114] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2005. Corr. 2nd printing.

[115] Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.

[116] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[117] Michael Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[118] Michael Norrish. Deterministic expressions in C. In S. Doaitse Swierstra, editor, *ESOP*, volume 1576 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 1999. doi: 10.1007/3-540-49099-X_10.

[119] Colin O'Halloran. Guess and verify – back to the future. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009, Proceedings*, volume 5850, pages 23–32. Springer, 2009. doi: 10.1007/978-3-642-05089-3_3.

[120] Joseph O'Rourke. *Computational Geometry in C.* Cambridge University Press, 2nd edition, 1998.

[121] H.-J. Ostermann. Neue Maschinenrichtlinie 2006/42/EG sowie Maschinenrichtlinie 98/37/EG. http://www.maschinenrichtlinie.de, 2009. Retrieved Oct 2009.

[122] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *CADE*, pages 748–752, 1992. doi: 10.1007/3-540-55602-8_217.

[123] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[124] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Jounal*, 19:53–77, 1997.

[125] Jan Peleska. Formal methods and the development of dependable systems. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Bericht Nr. 9612, December 1996. Habilitationsschrift.

[126] Jan Peleska. A unified approach to abstract interpretation, formal verification and testing of C/C++ modules. In John S. Fitzgerald, Anne E. Haxthausen, and Husnu Yenigun, editors, *Theoretical Aspects of Computing - ICTAC 2008*, volume 5160 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2008. doi: 10.1007/978-3-540-85762-4.

[127] Wojciech Penczek and Agata Polrola. *Advances in Verification of Time Petri Nets and Timed Automata.* Studies in Computational Intelligence. Springer, 2006. doi: 10.1007/978-3-540-32870-4.

[128] André Platzer and Edmund M. Clarke. Formal verification of curved flight collision avoidance maneuvers: A case study. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *LNCS*, pages 547–562. Springer, 2009. doi: 10.1007/978-3-642-05089-3_35.

[129] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977. doi: 10.1109/SFCS.1977.32.

[130] Hendrik Post and Wolfgang Küchlin. Integrated static analysis for Linux device driver verification. In *Integrated Formal Methods*, volume 4591 of *Lecture Notes in Computer Science*, pages 518–537. Springer, 2007. doi: 10.1007/978-3-540-73210-5_27.

[131] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 7:156–173, 2005.

[132] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Los Alamitos, California, 2002. IEEE Computer Society.

[133] Peter J. Robinson and John Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1):47–61, 1993. doi: 10.1093/logcom/3.1.47.

[134] RTCA. *DO-178B – Software considerations in airborne systems and equipment certification.* RTCA, Inc., Washington, D.C., United States, 1992. Errata March 1999.

[135] John Rushby. Formal methods and their role in the certification of critical systems. Technical report, Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop), 1995.

[136] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL.* PhD thesis, Technische Universität München, 2006.

[137] Norbert Schirmer and Makarius Wenzel. State spaces – The locale way. In *4th International Workshop on Systems Software Verification (SSV 2009)*, volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 161–179. Elsevier Science B.V., 2009. doi: 10.1016/j.entcs.2009.09.065.

[138] Bastian Schlich and Stefan Kowalewski. Model checking C source code for embedded systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(3):187–202, July 2009. doi: 10.1007/s10009-009-0106-5.

[139] Peter Sewell, , Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20 (01):71–122, 2010. doi: 10.1017/S0956796809990293.

[140] David J. Smith and Kenneth G. L. Simpson. *Functional Safety – A straightforward guide to applying IEC 61508 and related standards.* Elsevier, second edition, 2004.

[141] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall Int. Ltd., 1992.

[142] Steve Summit. comp.lang.c Frequently Asked Questions. http://c-faq.com, 2005. Retrieved Feb 2010.

[143] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics.* MIT Press, 2005.

[144] Harvey Tuch. *Formal Memory Models for Verifying C Systems Code.* PhD thesis, The University of New South Wales, 2008.

[145] Harvey Tuch. Formal verification of C systems code. *Journal of Automated Reasoning*, 42(2–4):125–187, April 2009. doi: 10.1007/s10817-009-9120-2.

[146] UK Health and Safety Executive. Reducing risks, protecting people. HSE Books, Sudbury, Suffolk, 2001.

[147] VDI Leitfaden. Leitfaden FTS-Sicherheit. VDI-Gesellschaft Fördertechnik Materialfluss Logistik – Fachbereich B7 Fahrerlose Transportsysteme (FTS), 2009.

[148] Verified Systems International GmbH. RT-Tester 6.0 – User Manual. `http://www.verified.de/en/products/rt-tester`, 2008. Retrieved Jul 2010.

[149] Dennis Walter, Lutz Schröder, and Till Mossakowski. Parametrized exceptions. In Jose Fiadeiro and Jan Rutten, editors, *Algebra and Coalgebra in Computer Science*, volume 3629 of *Lecture Notes in Computer Science*, pages 424–438. Springer, 2005. doi: 10.1007/11548133_27.

[150] Dennis Walter, Holger Täubig, and Christoph Lüth. Experiences in applying formal verification in robotics. In Erwin Schoitsch, editor, *29th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP 2010)*, volume 6351 of *Lecture Notes in Computer Science*, pages 347–360. Springer, 2010. doi: 10.1007/978-3-642-15651-9_26.

[151] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In *Automated Deduction – CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009. doi: 10.1007/978-3-642-02959-2_10.

[152] Markus Wenzel. *Isabelle/Isar – A versatile environment for human-readable formal proof documents.* PhD thesis, Institut für Informatik, Technische Universität München, 2002.

[153] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 349–361, New York, NY, USA, 2008. ACM. doi: 10.1145/1375581.1375624.