Karlsruhe Institute of Technology

Test, Validation, Analysis

Correctness-by-Construction for Pancake Programs

Master's Thesis
of

Jakob Jarebica

at the Department of Informatics
Institute of Information Security and Dependability
Test, Validation and Analysis (TVA)

Reviewer:	Prof. Dr.-Ing. Ina Schaefer
Second Reviewer:	Assoc. Prof. Dr. Alex Potanin
Advisor:	Maximilian Kodetzki, M.Sc.

Completion period: June 30, 2025 – February 6, 2026

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 6. Februar 2026 Jakob Jarebica

Abstract

In safety-critical environments, operating systems are commonly used. Therefore, they need to satisfy strict requirements on their functional correctness, making them a suitable target for the application of formal techniques, such as formal verification. Existing research on the verification of systems software, e.g., work on the *seL4* microkernel, shows that this process is time consuming and expensive. The systems programming language *Pancake* addresses this issue through its language design, featuring a verified compiler, based on a semantics formalized in the interactive theorem prover *HOL4*.

However, existing formal techniques for systems programming languages are limited to *post-hoc* approaches, i.e., the verification of a completed implementation against a specification. Since this specification is typically known before implementation, there is potential for improvement. In contrast to *post-hoc* approaches, refinement-based techniques like Correctness-by-Construction (CbC) exploit this knowledge by using the specification as the origin for correctness-preserving refinement steps, resulting in a program which fulfills the initial specification *a priori*. The refinement steps in CbC consist of the user-guided application of proven-correct refinement rules which simplify selecting correct refinement steps.

In this thesis, we introduce a CbC calculus for a Turing-complete subset of the systems programming language *Pancake* which allows developers to implement *Pancake* programs by refining a Hoare-style specification into programs using a set of 26 refinement rules. Our CbC calculus is implemented in *HOL4*, leveraging the formal semantics of *Pancake* to achieve provable end-to-end correctness. To demonstrate the feasibility of our approach, we present an implementation of linear search as a case study as well as an approach to automate the refinement proofs.

Kurzfassung

In sicherheitskritischen Umgebungen werden häufig Betriebssysteme eingesetzt. Daher müssen sie strikte Anforderungen an ihre funktionale Korrektheit erfüllen, was sie zu einem geeigneten Ziel für die Anwendung formaler Techniken macht, beispielsweise für formale Verifikation. Bestehende Forschungsarbeiten, zum Beispiel die Arbeiten zum *seL4*-Mikrokernel, zeigen, dass die Verifikation von Systemsoftware zeitaufwändig und kostspielig ist. Die Systemprogrammiersprache *Pancake* geht dieses Problem durch ihr Sprachdesign an, das einen verifizierten Compiler umfasst, der auf einer im interaktiven Theorembeweiser *HOL4* formalisierten Semantik basiert.

Die bestehenden formalen Techniken für Systemprogrammiersprachen beschränken sich jedoch auf *Post-Hoc-Ansätze*, das heißt die Verifikation einer abgeschlossenen Implementierung gegen eine Spezifikation. Da diese Spezifikation in der Regel bereits vor der Implementierung bekannt ist, besteht hier Verbesserungspotenzial. Im Gegensatz zu *Post-Hoc-Ansätzen* nutzen Refinement-basierte Techniken wie Correctness-by-Construction (CbC) dieses Wissen, indem sie die Spezifikation als Ausgangspunkt für korrektheitserhaltende Refinement-Schritte verwenden, was zu einem Programm führt, das die ursprüngliche Spezifikation *a priori* erfüllt. Die Refinement-Schritte in CbC bestehen in der benutzergeführten Anwendung formal bewiesener Refinement-Regeln, welche die Auswahl korrekter Refinement-Schritte vereinfachen.

In dieser Arbeit stellen wir einen CbC-Kalkül für eine Turing-vollständige Teilmenge der Systemprogrammiersprache *Pancake* vor, der es Entwicklern ermöglicht, Pancake-Programme zu implementieren, indem sie eine Hoare-Spezifikation unter Verwendung von 26 Refinement-Regeln zu einem Pancake-Programm zu verfeinern. Unser CbC-Kalkül ist in *HOL4* implementiert und nutzt die formale Semantik von *Pancake*, um beweisbare Ende-zu-Ende-Korrektheit zu erreichen. Um die Machbarkeit unseres Ansatzes zu zeigen, präsentieren wir eine Implementierung der linearen Suche als Fallstudie sowie einen Ansatz zur Automatisierung der Refinement-Beweise.

Contents

List of Figures

Abbreviations

CbC	Correctness-by-Construction
HOL	higher order logic
GCL	Guarded Command Language
LEM	law of excluded middle
SIL	safety integrity level
SML	Standard ML
WP	weakest precondition

1. Introduction

Operating systems are prevalent in most safety-critical environments. The applicable standards for software quality assurance, especially IEC 61508 [Int10], highly recommend the usage of formal techniques for applications that require a elevated safety integrity level (SIL), i.e., SIL 3 or 4. One of the most common formal techniques is *formal verification*, i.e., proving that a given implementation fulfills a given specification of functionality. Existing work on the formal verification of low-level software, e.g., the work on the seL4 microkernel [Kle+14], shows that the verification of low-level software implemented in common systems languages, i.e., C and Rust, is time consuming and thus expensive.

Pohjola et al. [Poh+23] approach this issue focusing on the programming language design, introducing the systems language *Pancake*. Pancake is a programming language at a level of abstraction between C and assembly which eliminates undesirable properties from C, e.g., the complex memory model while being sufficiently expressive for systems programming. The semantics of Pancake is defined formally in higher order logic (HOL), allowing Pohjola et al. [Poh+23] to prove their Pancake compiler correct using the interactive theorem prover *HOL4*. Ongoing work on formal verifiers for Pancake code aims to enable the development of proven-correct Pancake software, providing provable correctness from source code down to binaries using the verified compiler.

Additional potential to simplify the application of formal methods to systems software lies in the choice of the techniques: formal verification is a *post-hoc* technique, i.e., software is verified against a given specification *after* it is implemented. This leaves room for improvement, as the desired functionality is usually known before implementation. Using a refinement-based process can exploit this knowledge to allow the implementation of software that is formally correct *a priori*. There are multiple advancements in defining an approach based on refining a formal specification of functionality into software that fulfills this specification. This type of approach is initially described by Dijkstra [Dij68] and advanced by Morgan [Mor94]. A modern approach in this direction is Correctness-by-Construction (CbC) [KW12]. In CbC, a formal specification of functionality is given as a Hoare triple $\{P\}\ S\ \{Q\}$, where P and Q are precondition and postcondition, both given in first-order logic, and S is a statement placeholder. CbC defines a set of refinement rules which specify how a root Hoare triple can be refined into one or multiple Hoare triples or program statements. Correctly applying the refinement rules repetitively results in an implementation in Dijkstra's *Guarded Command Language* (GCL) [Dij75] that fulfills the initial specification. The set of refinement rules within the CbC calculus is proven sound and complete for terminating programs. CbC forces the developer to make formally correct refinements; still, it is not an automated process comparable to program synthesis. The decision of which refinement rule to use is left to the developer who remains responsible for all design decisions during the implementation.

The existing CbC processes are designed for high-level programming languages and cannot be easily transferred to low-level programming languages, as the CbC refinement rules do not resemble some constructs of low-level programs, especially the access to addressable memory. Additionally, the current CbC implementations rely on Java-specific verifiers [Bor+23]. Although using specialized systems programming languages, e.g., Pancake, facilitates the formal verification of low-level programs, the existing approaches are limited to *post-hoc* approaches.

To close this gap, this thesis introduces a CbC process for the implementation of programs in a subset of the programming language *Pancake*. Our CbC process allows the developer to refine a formal specification of the desired functionality into Pancake code that fulfills the functionality described by the formal specification. It is implemented and proven in the interactive theorem prover *HOL4* which allows us to utilize the formal semantics of Pancake, ensuring the end-to-end correctness of our calculus. This extends the CbC paradigm to the domain of systems programming.

The remainder of this thesis is structured as follows: in Chapter 2, we introduce the foundations of this thesis, focussing on the origins and state of the art in CbC and providing an introduction into Pancake and HOL4. In Chapter 3, we describe our CbC calculus for Pancake programs, including a formalization of Hoare triples for Pancake, weakest preconditions for Pancake statements, the definition of contract refinement, and refinement rules. To demonstrate the feasibility of our approach, we provide a case study in Chapter 4, in which we develop a Pancake implementation of linear search from its specification using our CbC calculus. In Chapter 5, we give an overview about existing research related to this thesis. In Chapter 6, we conclude this thesis by drawing an conclusion and giving an outlook to future work.

2. Foundations

In this chapter, we introduce into the foundations of our contributions which are documented in Chapter 3. In Section 2.1, we describe the development paradigm Correctness-by-Construction (CbC), embedded in its historical context and using CbC for Dijkstra's Guarded Command Language (GCL) as an example. In Section 2.2, we provide an introduction into the programming language *Pancake*, focussing on its semantics. In Section 2.3, we explain the basics of goal-directed proofs in *HOL4*.

2.1 Correctness-by-Construction

Correctness-by-Construction (CbC) is a development paradigm. A developer using CbC begins by providing a formal specification of a required functionality, usually as Hoare-style preconditions and postconditions. Following that, the developer uses a set of proven-correct refinement rules to transform this initial specification. Each refinement rule introduces a construct of the used programming languages, e.g., a selection or a repetition statement, leaving new specifications to refine to create their statement bodies. Using the right refinement rules, this step-wise refinement ultimately results in a concrete program which implements the required functionality.

This type of development approach, beginning with a specification and transforming it into a program, is called an *a priori* approach. *A priori* approaches to program correctness were first proposed by Dijkstra in 1968 who describes them as being similar to the behavior of a developer. Dijkstra does not provide a detailed description of a process, but describes his ideas on a complex example, and concludes:

> I have not done much more than to make explicit what the competent programmer has already done for years, be it mostly intuitively and unconsciously. [Dij68]

In 1981, Back makes the first steps towards an *a priori* approach that uses formal methods by proving a refinement calculus for programs in the Guarded Command Language (GCL) [Dij75; Bac81]. This calculus is enhanced in the following years

[Bac88; BW90] and used by Morgan in 1994 to formalize an *a priori* approach for GCL programs [Mor94].

More recently, Kourie and Watson rephrase these ideas using a more modern terminology, calling the development paradigm Correctness-by-Construction (CbC) [KW12]. Modern style CbC is similar to the approach described by Morgan, but focusses more strongly on the developer's perspective, simplifying its application by casting the allowed refinements into simple rules.

To facilitate the use of these approaches, there is tool support: The calculus by Back has been implemented multiple times [BL96; Car+98]. Kourie's and Watson's CbC has been implemented by Runge et al. in the graphical tool CorC [Run+19].

This thesis is based on the concepts developed by the authors mentioned above. The contributions of Morgan, Kourie, and Watson are the closest to our own work. Therefore, we explain their terminology and concepts in the remainder of this section, using CbC for GCL as a running example.

2.1.1 Programs, Specifications, and Contracts

For introducing CbC as a development paradigm, we initially have to specify which kind of product is developed using CbC. Commonly, the resulting products of software development processes are called *programs*, but it seems relativelty complicated to describe what the term *program* exactly refers to. The most obvious definition matches our colloquial use of the term *program* and can be found in ISO/IEC 2382-1 [II15]:

Definition 2.1 (Program – ISO/IEC 2382-1)**.** *Syntactic unit that conforms to the rules of a particular programming language and that is composed of declarations and statements or instructions needed to solve a certain function, task, or problem.*

This definition hints at two aspects to initially address for our running example, CbC for GCL: (1) the *syntactic rules* for GCL programs and (2) in which cases a *program* solves a task. In other words, we have to define both (1) the syntax of the programming language and (2) a way to specify functionality. Beginning with the first aspect, we define the syntax of GCL as follows:

Definition 2.2 (GCL Programs)**.** *In this thesis, we use a subset of Dijkstra's original GCL [Dij75]. The syntax of this subset is given as a BNF grammar in Figure 2.1. GCL* programs *are statements that can be constructed by this grammar.*

To address the second aspect of *programs*, specify their functionality, we also consult ISO/IEC 2382-1 [II15] which defines *program specification* as follows:

Definition 2.3 (Program Specification – ISO/IEC 2382-1)**.** *Document that describes the structure and functions of a program in sufficient detail to permit programming and to facilitate maintenance.*

According to this definition, *specifications* can be given in different forms: a document in a natural language that is sufficiently detailed can permit programming, a *formal specification* can as well. The difference between both is the used notation and the resulting difference in ambiguity as ISO/IEC 2382-1 [II15] hints at:

Expr	e	$::=$	v	variable
		$\mid$	c	integer constant
		$\mid$	$e + e$	addition
		$\mid$	$e - e$	subtraction
		$\mid$	$e = e$	equal comparison
		$\mid$	$e < e$	less-than comparison
		$\mid$	$e \leq e$	less-or-equal comparison
Stmt	s	$::=$	Skip	empty statement
		$\mid$	$v\ :=\ e$	variable assignment
		$\mid$	$s;\ s$	sequence statement
		$\mid$	If $\{\ b\ \}$	selection statement
		$\mid$	While $\{\ b\ \}$	repetition statement
Body	b	$::=$	$e \rightarrow s$	guarded command
		$\mid$	$e \rightarrow s,\ b$	guarded command list

Figure 2.1: Syntax of GCL

Definition 2.4 (Formal Specification – ISO/IEC 2382-1)**.** *Specification written in a formal notation, often for use in correctness proving.*

For CbC, we require *formal specifications*, as we want to achieve provable correctness. One kind of *formal specification* are Hoare-style precondition and postcondition pairs. For this thesis, we use Hoare triples as specifications of *total correctness*, i.e., requiring termination, and define them as follows:

Definition 2.5 (Hoare Triple)**.** *A program S fulfills the specification given as the precondition P and postcondition Q iff executing the program S terminates in a state fulfilling Q when executed from any initial state fulfilling P. If this holds, we say that the* Hoare triple $\{P\}\ S\ \{Q\}$ *holds.*

Applying this definition to our running example, GCL, we first have to define a semantics for the language, i.e., what executing a *program* means. For the sake of simplicity, we generally assume the standard semantics of GCL [Dij75], but make the following two adaptions: (1) We assume that all programs terminate. A more complex semantics that allows diverging programs is demonstrated in Appendix B using the interactive theorem prover *HOL4*. (2) For guarded commands, we make the simplification to always select the first matching guard, allowing deterministic evaluation. In combination, both adaptions allow us to assume a total and deterministic evaluation function for GCL statements. Given this semantics, we define Hoare triples for GCL as follows:

Definition 2.6 (Hoare Triples for GCL)**.** *The program states s of a GCL program consists of the values of its variables, i.e., s : varname $\longrightarrow$ int. Non-assigned variables are assumed as zero-initialized. Preconditions P and postconditions Q are both sets of program states, specifying which states are legal.*

For a given GCL statement stmt, $\{P\}$ stmt $\{Q\}$ holds iff

$$\forall s : s \in P \Rightarrow \mathit{evaluate}(\mathit{stmt}, s) \in Q,$$

where evaluate is the total evaluation function on statements.

Having defined GCL *programs* and Hoare triples for GCL as *formal specifications*, we can continue specifying CbC for GCL. In the terms defined above, CbC is a development paradigm in which a *formal specification*, usually a Hoare specification, is refined into a *program* by step-wise application of refinement rules. However, there are intermediate products created in the process: After the application of one refinement rule to a initial *formal specification*, the developer usually does not receive a *program*, rather an object that formalizes requirements to the syntax of the *program*, but leaves some *gaps* in the syntax, only specifying the semantics of these *gaps*. We can demonstrate this in the following example for the GCL sequence statement:

Example 2.1 (Intermediate Products)**.** *Given a refinement rule that allows the refinement of a Hoare specification of P and Q towards a GCL program containing a sequence statement s_1; s_2, the smallest possible refinement step would be introducing the sequence itself, but leaving s_1 and s_2 as gaps. To preserve correctness, the rule would enforce $\{P\}\ s_1\ \{M\}$ and $\{M\}\ s_2\ \{Q\}$ for some intermediate condition M.*

We could specify the full rule as follows, denoting gaps as $(\cdot)$:

$$\{P\} \cdot \{Q\} \sqsubseteq \{P\} \cdot \{M\};\ \{M\} \cdot \{Q\}$$

The term $\{P\} \cdot \{M\};\ \{M\} \cdot \{Q\}$ in the example above is neither a *formal specification* in the usual sense, as it talks about program syntax, nor is it a *program*, as it contains indirect specification of semantics. Morgan calls these terms *sub-specifications*, alternatively *super-programs*, and proposes to subsume all three, programs, specifications, and sub-specifications, under the term *program* [Mor94]. This is justified by the fact that we treat them quite similar in refinement-based techniques, as all of them specify functionality in some way. On the other hand, this terminology complicates talking about *programs* and *specifications*, each in the sense of ISO/IEC 2382-1 [II15], requiring the usage of longer terms like *concrete program* and *abstract program*.

To keep the terminology in this thesis simple, we will use the terms (1) *program* and (2) *specification* in the sense of ISO/IEC 2382-1 [II15], the term (3) *sub-specification* in the sense of Morgan [Mor94], and subsume all of them under the term *contract* which is also used by Morgan.

Programs can satisfy contracts, informally defining:

Definition 2.7 (Satisfication of Contracts)**.** *Given a set of statements Stmt and a set of contracts $\mathsf{Contract}$, we can define a relation $\mathsf{sat} \subseteq \mathsf{Stmt} \times \mathsf{Contract}$ that specifies which statements satisfy which contracts.*

A program satisifies

1. *itself and no other program,*
2. *a Hoare specification if the Hoare triple as defined above holds, and*
3. *a sub-specification if it fulfills all syntactic and semantic requirements.*

We can define *contracts* for the refinement-based development in GCL and their satisfication as follows:

Definition 2.8 (Contracts for GCL)**.** *The contracts to be used for refinement-based programming in GCL are constructed by the BNF grammar below. The symbols s and e denote statements and expressions, defined in Figure 2.1.*

$$\begin{array}{llcll}
\textsf{Contract} & c & ::= & \textsf{HoareC}\ P\ Q & \textit{hoare contract} \\
 & & \mid & \textsf{SeqC}\ c\ c & \textit{sequence contract} \\
 & & \mid & \textsf{IfC}\ b & \textit{selection contract} \\
 & & \mid & \textsf{WhileC}\ b & \textit{repetition contract} \\
 & & \mid & \textsf{ProgC}\ s & \textit{program contract} \\
\textsf{Body} & b & ::= & e \to c & \textit{guarded contract} \\
 & & \mid & e \to c\texttt{,}\ b & \textit{guarded contract list}
\end{array}$$

Let $[e_i \to p_i]_1^k$ be a shorthand notation for a list of guarded commands or guarded contracts. We define the satisfication relation sat *as follows:*

$$\begin{array}{lcl}
\textsf{stmt sat}\ (\textsf{HoareC}\ P\ Q) & \iff & \forall s : s \in P \Rightarrow \textsf{evaluate}(\textsf{stmt}, s) \in Q \\
\textsf{stmt sat}\ (\textsf{SeqC}\ c_1\ c_2) & \iff & \exists p_1, p_2 : \textsf{stmt} = (\textsf{Seq}\ p_1\ p_2) \wedge \\
 & & p_1\ \textsf{sat}\ c_1 \wedge p_2\ \textsf{sat}\ c_2 \\
\textsf{stmt sat}\ (\textsf{IfC}\ [e_i \to c_i]_1^k) & \iff & \exists p_1, \ldots, p_k : \textsf{stmt} = (\textsf{If}\ [e_i \to p_i]_1^k) \wedge \\
 & & \forall i, 1 \leq i \leq k : p_i\ \textsf{sat}\ c_i \\
\textsf{stmt sat}\ (\textsf{WhileC}\ [e_i \to c_i]_1^k) & \iff & \exists p_1, \ldots, p_k : \textsf{stmt} = (\textsf{While}\ [e_i \to p_i]_1^k) \wedge \\
 & & \forall i, 1 \leq i \leq k : p_i\ \textsf{sat}\ c_i \\
\textsf{stmt sat}\ (\textsf{ProgC}\ p) & \iff & \textsf{stmt} = p
\end{array}$$

2.1.2 Refinement of Contracts

As discussed in the previous section, the objects *refined* into each other in a CbC process are *contracts*. Following the definition by Kourie and Watson [KW12], we define the refinement relation on contracts and prove some basic properties:

Definition 2.9 (Refinement of Contracts). *Let* $\textsf{sat} \subseteq \textsf{Stmt} \times \textsf{Contract}$ *be the satisfication relation between programs and contracts, i.e.,* p sat c *iff* p *satisfies* c.

A contract c_1 *can be refined into a contract* c_2*, denoted by* $c_1 \sqsubseteq c_2$*, iff*

$$\forall p : p\ \textsf{sat}\ c_2 \Rightarrow p\ \textsf{sat}\ c_1,$$

i.e., that all programs satisfying c_2 *also satisfy* c_1.

Theorem 2.1 (Reflexivity and Transitivity of Refinement). *The refinement relation* $\sqsubseteq$ *is reflexive and transitive, i.e.,*

$$\begin{array}{lcl}
\forall c & : & c \sqsubseteq c \\
\forall c_1, c_2, c_3 & : & c_1 \sqsubseteq c_2 \wedge c_2 \sqsubseteq c_3 \Rightarrow c_1 \sqsubseteq c_3
\end{array}$$

Proof. Let c_1, c_2, c_3 be contracts and p be a program.

For reflexivity: p sat $c_1 \Rightarrow p$ sat c_1 holds trivially, and thus $c_1 \sqsubseteq c_1$ per Definition 2.9.

For transitivity: Let $c_1 \sqsubseteq c_2$ and $c_2 \sqsubseteq c_3$. According to Definition 2.9, p sat $c_3 \Rightarrow p$ sat c_2 and p sat $c_2 \Rightarrow p$ sat c_1 hold. By transitivity of the implication, p sat $c_3 \Rightarrow p$ sat c_1, and therefore, $c_1 \sqsubseteq c_3$. □

Corollary 2.2 (Refinement from Specification to Programs). *Let* c_1 *be a specification,* $c_2, \ldots, c_{k-1}$ *be sub-specifications, and* c_k *be a program. By transitivity,*

$$\forall i, 1 \leq 1 \leq k - 1 : c_i \sqsubseteq c_{i+1}$$

implies that $c_1 \sqsubseteq c_k$*, i.e., programs obtained by multiple refinement steps from a specification fulfill that specification.*

As proven above, the refinement relation is reflexive and transitive without knowledge of the satisfication relation. Knowing the language-specific definition of the satisfication relation sat, we can prove another property of GCL contract refinement often used in practice:

Theorem 2.3 (Monotonicity of GCL Contract Refinement)**.** *The composition of contracts is monotonic with respect to the refinement relation, i.e., given* $c_j \sqsubseteq c_j^*$*, it holds that*

$$\begin{array}{lcl} \mathit{SeqC}\ c_j\ c & \sqsubseteq & \mathit{SeqC}\ c_j^*\ c \\ \mathit{SeqC}\ c\ c_j & \sqsubseteq & \mathit{SeqC}\ c\ c_j^* \\ \mathit{IfC}\ [e_i \rightarrow c_i]_1^k & \sqsubseteq & \mathit{IfC}\ [e_i \rightarrow c_i]_1^k \setminus [e_j \rightarrow c_j^*]_j \\ \mathit{WhileC}\ [e_i \rightarrow c_i]_1^k & \sqsubseteq & \mathit{WhileC}\ [e_i \rightarrow c_i]_1^k \setminus [e_j \rightarrow c_j^*]_j, \end{array}$$

where $[e_i \rightarrow c_i]_1^k \setminus [e_j \rightarrow c_j^*]_j$ *denotes a list of guarded contracts, in which* c_j *is replaced with* c_j^* *at index* j*.*

Proof. In Theorem B.11 in Appendix B, we prove a corresponding theorem for the GCL semantics that allows diverging programs. The proof of the theorem above is a special case of the proof developed using *HOL4* in Appendix B and therefore omitted for brevity. ◊

2.1.3 Refinement Rules

The definitions and theorems presented in the preceding section lay the formal foundations of CbC, but are difficult to use for a specification-to-program refinement proof: Although it is possible to prove, e.g., a refinement of a initial specification into a sub-specification correct, the definitions above do not help choosing that sub-specification. Additionally, the correctness proofs of a refinement given only the definitions above is usually rather intricate, as every single refinement proof has to deal with the semantic properties of the evaluation function.

To simplify the use of refinement techniques, we capture this complexity in *refinement rules*. Refinement rules already occur in Morgan's contributions; he lists more than 70 rules in the appendix to his work [Mor94]. Kourie and Watson state that most of these rules are more of a theoretical interest than of pracical relevance, limiting the refinement rules for modern-style CbC to a smaller number of rules [KW12]. Their refinement rules follow a common pattern which we present in following definition:

Definition 2.10 (CbC Refinement Rules)**.** *Refinement rules for modern-style CbC formalize the introduction of a statement type, the strengthening of a postcondition, or the weakening of a precondition.*

Before applying a CbC refinement rule to a specification, the developer has to prove that the precondition implies the rule's side conditions*. The application of a CbC rule leads to a contract, i.e., a specification, a sub-specification, or a program, that can be the starting point of further refinement steps. Therefore, CbC refinement rules share the following pattern:*

$$(\mathit{side\ conditions}) \Rightarrow (\mathit{specification}) \sqsubseteq (\mathit{contract})$$

If a rule has no side conditions, the implication is omitted.

To visualize this definition, we list the refinement rules by Kourie and Watson for GCL [KW12].

Example 2.2 (Refinement Rules for GCL)**.** *Let* $\Rrightarrow$ *the* implies everywhere *relation on conditions, i.e.,*

$$P \Rrightarrow Q \Longleftrightarrow \forall s : s \in P \Rightarrow s \in Q,$$

G_i *the condition equivalent to the expression* e_i *evaluating to a greater-zero value, and* $GG = G_1 \vee \cdots \vee G_k$ *the disjunction of all* G_i*.*

The following refinement rules hold for GCL:

1. *Strengthen Postcondition Rule:*
   $(Q' \Rrightarrow Q) \Rightarrow (\mathsf{HoareC}\ P\ Q) \sqsubseteq (\mathsf{HoareC}\ P\ Q')$
2. *Weaken Precondition Rule:*
   $(P \Rrightarrow P') \Rightarrow (\mathsf{HoareC}\ P\ Q) \sqsubseteq (\mathsf{HoareC}\ P'\ Q)$
3. *Skip Rule:*
   $(P \Rrightarrow Q) \Rightarrow (\mathsf{HoareC}\ P\ Q) \sqsubseteq (\mathsf{ProgC}\ \mathsf{Skip})$
4. *Assignment Rule:*
   $(P \Rrightarrow Q[v \setminus e]) \Rightarrow (\mathsf{HoareC}\ P\ Q) \sqsubseteq (\mathsf{ProgC}\ (\mathsf{Assign}\ v\ e))$
5. *Sequence Rule:*
   $(\mathsf{HoareC}\ P\ Q) \sqsubseteq (\mathsf{SeqC}\ (\mathsf{HoareC}\ P\ M)\ (\mathsf{HoareC}\ M\ Q))$
6. *Selection Rule:*
   $(P \Rrightarrow GG) \Rightarrow (\mathsf{HoareC}\ P\ Q) \sqsubseteq (\mathsf{IfC}\ [e_i \rightarrow (\mathsf{HoareC}\ (P \wedge G_i)\ Q)]_1^k)$
7. *Repetition Rule (without termination):*
   $(P \Rrightarrow I \wedge I \Rrightarrow \neg GG) \Rightarrow (\mathsf{HoareC}\ P\ Q) \sqsubseteq (\mathsf{WhileC}\ [e_i \rightarrow (\mathsf{HoareC}\ (I \wedge G_i)\ I)]_1^k)$

In the following, we exemplarily prove the *sequence rule.* The remaining rules are omitted for brevity and proven in Appendix B using *HOL4*.

Theorem 2.4 (GCL Sequence Rule)**.** *The refinement rule*

$$\mathsf{HoareC}\ P\ Q \sqsubseteq \mathsf{SeqC}\ (\mathsf{HoareC}\ P\ M)\ (\mathsf{HoareC}\ M\ Q)$$

holds.

Proof. By applying the definitions we present above, it holds that

$$\begin{aligned}
& & & \mathsf{HoareC}\ P\ Q \sqsubseteq \mathsf{SeqC}\ (\mathsf{HoareC}\ P\ M)\ (\mathsf{HoareC}\ M\ Q) \\
\overset{\text{def 2.9}}{\Longleftrightarrow} & & \forall \mathsf{s} : \ & \mathsf{s\ sat}\ (\mathsf{SeqC}\ (\mathsf{HoareC}\ P\ M)\ (\mathsf{HoareC}\ M\ Q)) \Rightarrow \\
& & & \mathsf{s\ sat}\ (\mathsf{HoareC}\ P\ Q) \\
\overset{\text{def 2.8}}{\Longleftrightarrow} & & \forall p_1, p_2 : \ & p_1\ \mathsf{sat}\ (\mathsf{HoareC}\ P\ M) \wedge \\
& & & p_2\ \mathsf{sat}\ (\mathsf{HoareC}\ M\ Q) \Rightarrow \\
& & & (\mathsf{Seq}\ p_1\ p_2)\ \mathsf{sat}\ (\mathsf{HoareC}\ P\ Q) \\
\overset{\text{def 2.8}}{\Longleftrightarrow} & & \forall p_1, p_2 : \ & (\forall t : t \in P \Rightarrow \mathsf{evaluate}(p_1, t) \in M) \wedge \\
& & & (\forall t : t \in M \Rightarrow \mathsf{evaluate}(p_2, t) \in Q) \ \Rightarrow \\
& & & (\forall t : t \in P \Rightarrow \mathsf{evaluate}(\mathsf{Seq}\ p_1\ p_2, t) \in Q).
\end{aligned}$$

The consequent of this implication requires that evaluating a sequence statement $\mathsf{Seq}\ p_1\ p_2$ from an initial state $s \in P$ results in a state $t \in Q$. Assuming the GCL semantics discussed in Section 2.1.1, evaluating a sequence statement Seq p_1 p_2 from an initial state $s \in P$ means (1) first evaluating p_1 from s resulting in a state t_1, (2) then evaluating p_2 from t_1 resulting in a state t_2, requiring $t_2 \in Q$. By the first conjunct of the antecedent, it holds that $s \in P$ implies $t_1 \in M$, and by the second conjunct, $t_1 \in M$ implies $t_2 \in Q$. $\square$

2.2 Pancake

Pancake is a systems programming language introduced by Pohjola et al. [Poh+23] which is part of the *CakeML* ecosystem [Kum+14]. Pancake is targeted to improve the security of operating systems by simplifying the verification of device drivers, which is reflected in the language's syntax and semantics.

Pancake features an abstraction level between C and assembly and a minimal type system, only supporting *machine words* and *nestable structs* of machine words. In Pancake, these types are called *shapes*. Local variables are stack-allocated, values may also be saved and loaded from global variables and a statically allocated heap. We present the subset of the Pancake syntax we support in this thesis in Section 2.2.1.

The developers of Pancake formalize the language's semantics as *functional big-step semantics* [Owe+16], embedded in a theory for the theorem prover *HOL4* [SN08]. The Pancake compiler is proven correct and uses parts of the compiler stack of *CakeML* [Kum+14]. We explain the most relevant properties of Pancake's semantics in Section 2.2.2 and the very basics of proofs in *HOL4* in Section 2.3.

As Pancake is a programming language under ongoing development, we decided to freeze the Pancake version used for this thesis at the Git commit `721c4576e` of *CakeML*[1]. This was the up-to-date version when finalizing our refinement calculus proofs. For *HOL4*, we use the release version *Trindemossen 2*[2].

2.2.1 Syntax

For this thesis, we decided to limit ourselves to a Turing-complete subset of Pancake, leaving function calls, the foreign function interface, and shared memory operations for future work. The abstract syntax of this language subset is shown in Figure 2.2, where the algebraic data types *exp* and *prog* represent Pancake expressions and statements. The type variable α stands for the word size of the hardware platform.

We illustrate the concrete syntax of Pancake with the example in Figure 2.3 where we implement a linear search on the heap. The example especially shows the concrete syntax of Pancake *shapes*: Function arguments are statically typed by prefixing their shape, e.g., declares `2 array` a two-element struct. The contents of structs can be accessed as, e.g., `array.1` and `array.2`. The load shape statement `lds` requires the static specification of the shape to fetch.

For the remainder of this thesis, we focus on HOL4 proofs about language properties and thus on the abstract syntax, making the following definition:

Definition 2.11 (Pancake Program). *A Pancake program is a HOL4 term of the type* α `prog`*, shown in Figure 2.2.*

[1]https://github.com/CakeML/cakeml
[2]https://github.com/HOL-Theorem-Prover/HOL

```
α exp =                          α prog =
Const (α word)                   Skip
| Var varkind mlstring           | Dec mlstring shape (α exp) (α prog)
| Struct (α exp list)            | Assign varkind mlstring (α exp)
| Field num (α exp)              | Store (α exp) (α exp)
| Load shape (α exp)             | Store32 (α exp) (α exp)
| Load32 (α exp)                 | StoreByte (α exp) (α exp)
| LoadByte (α exp)               | Seq (α prog) (α prog)
| Op binop (α exp list)          | If (α exp) (α prog) (α prog)
| Panop panop (α exp list)       | While (α exp) (α prog)
| Cmp cmp (α exp) (α exp)        | Break
| Shift shift (α exp) num        | Continue
| BaseAddr                       | Raise mlstring (α exp)
| TopAddr                        | Return (α exp)
| BytesInWord                    | Annot mlstring mlstring
```

Figure 2.2: Abstract Syntax of Pancake

2.2.2 Semantics

The semantics of pancake is formally defined as a *functional big-step semantics* [Owe+16], i.e., Pohjola et al. provide an evaluation function which yields a result and successor state for each program and initial state [Poh+23]. The evaluation results of Pancake are of the type α `result option` where a value of `NONE` represents normal termination and values of `SOME` r represent different reasons for abnormal termination.

To ensure termination of the evaluation function, the program state is equipped with a clock variable which is decremented in every loop iteration. If the clock value reaches zero, evaluation is terminated, yielding a result of `SOME TimeOut`. This behavior can be retraced by studying the record data type for the program state and the evaluation function for while loops, both presented in Figure 2.4 and Figure 2.5. The complete definition of `evaluate` is omitted here and can be found in Appendix C.

For use in the later chapters of this thesis, we formally define:

Definition 2.12 (Pancake Program State)**.** *A Pancake program state is a HOL4 term of the type* `state`*, shown in Figure 2.4.*

Definition 2.13 (Termination of Pancake Programs)**.** *A Pancake program prog is said to terminate when evaluated from a state s iff*

```
∃k. (let
       (r,t) = evaluate (p,s with clock := k)
     in
       r ≠ SOME TimeOut)
```

holds, i.e., if there is an initial clock value, such that evaluation does not time out. Otherwise, the Pancake program is said to diverge.

In addition to these definitions, Pohjola et al. provide a large set of proofs about properties of Pancake's semantics [Poh+23]. As we will use some of these theorems for our own proofs, we introduce the most relevant ones in the remainder of this section. The proofs of all following theorems are completed in HOL4 and therefore omitted here.

```
// Linear Search
// Arguments:
//   to_find: word to search for
//   array:   struct <start_address,length> to search in
// Returns:
//   Index of first occurrence in array, -1 if none
fun search(1 to_find, 2 array) {
    var curr_index = 0;

    // iterate while index is lower than array length
    while (curr_index < array.2) {

      // load word (shape 1) from current address
      var curr_value = lds 1 (array.1 + curr_index);

      if (curr_value == to_find) {
          return curr_index;
      }

      curr_index = curr_index + 1;
    }

    return -1;
}
```

Figure 2.3: Concrete Syntax of Pancake: Linear Search

Theorem 2.5 (Evaluation of Expressions is Clock-Independent). *For any expression e, program state s, and clock value ck, it holds that*

$\texttt{eval}\ (s\ \texttt{with}\ clock\ \texttt{:=}\ ck)\ e\ \texttt{=}\ \texttt{eval}\ s\ e,$

i.e., the evaluation of Pancake expressions is clock-independent.

Theorem 2.6 (Increasing the Clock when Evaluating Programs). *For any program p, program states s and t, results r, and clock increasements ck, it holds that*

$\texttt{evaluate}\ (p,s)\ \texttt{=}\ (r,t) \wedge r \neq \textit{SOME TimeOut} \Rightarrow$
$\texttt{evaluate}\ (p,s\ \texttt{with}\ clock\ \texttt{:=}\ s.clock + ck)\ \texttt{=}$
$(r,t\ \texttt{with}\ clock\ \texttt{:=}\ t.clock + ck),$

i.e., if a program terminates with one initial clock value, it terminates with any greater initial clock value in the same resulting state, and the clock increase propagates through to the resulting state.

Theorem 2.7 (Minimal Initial Clock Value for Terminating Programs). *For any program p, program states s and t, results r, and clock increasements ck, it holds that*

$\texttt{evaluate}\ (prog,s)\ \texttt{=}\ (r,t) \wedge r \neq \textit{SOME TimeOut} \Rightarrow$
$\exists k.\ \texttt{evaluate}\ (prog,s\ \texttt{with}\ clock\ \texttt{:=}\ k)\ \texttt{=}$
$\quad (r,t\ \texttt{with}\ clock\ \texttt{:=}\ 0),$

i.e., there exists a minimal initial clock value for any terminating program.

```
α state = <|
   locals : mlstring ↦ α v;
   globals : mlstring ↦ α v;
   eshapes : mlstring ↦ shape;
   memory : α word → α word_lab;
   memaddrs : α word → bool;
   clock : num;
   be : bool;
   base_addr : α word;
   top_addr : α word
|>
```

Figure 2.4: Semantics of Pancake: Program State

```
⊢ evaluate (While e c,s) =
  case eval s e of
    NONE ⇒ (SOME Error,s)
  | SOME (ValWord w) ⇒
    if w ≠ 0w then
      if s.clock = 0 then (SOME TimeOut,empty_locals s)
      else
        (let
           (r,s₁) = evaluate (c,dec_clock s)
         in
           case r of
             NONE ⇒ evaluate (While e c,s₁)
           | SOME Error ⇒ (r,s₁)
           | SOME TimeOut ⇒ (r,s₁)
           | SOME Break ⇒ (NONE,s₁)
           | SOME Continue ⇒ evaluate (While e c,s₁)
           | SOME (Return v₅) ⇒ (r,s₁)
           | SOME (Exception v₆ v₇) ⇒ (r,s₁))
    else (NONE,s)
  | SOME (Struct v₉) ⇒ (SOME Error,s)
```

Figure 2.5: Semantics of Pancake: Evaluation of While Loop

2.3 Goal-Directed Proofs in HOL4

HOL4 is an interactive theorem prover [SN08]. It allows specifications and proofs in classical HOL and is implemented in Standard ML (SML). In this section, we introduce the basics of *goal-directed* proofs in HOL4. For a more systematic introduction, we refer the interested reader to the overview paper by Slind and Norrish [SN08] and the HOL4 DESCRIPTION manual[3].

[3]https://hol-theorem-prover.org/#doc.

Notations

HOL4 allows the user to use two different notations: ASCII and Unicode. Editors for HOL4 support converting ASCII keyboard input to Unicode. Thus, HOL4 files usually contain the Unicode notation.

For the sake of introduction, we use the ASCII notation within this section and pretty-printed Unicode in the remainder of this thesis. The most common notations are displayed in Figure 2.6.

`~`	$\neg$	Boolean Negation
`/\`	$\wedge$	Boolean And
`\/`	$\vee$	Boolean Or
`==>`	$\Rightarrow$	Boolean Implication
`<=>`	$\Leftrightarrow$	Boolean Equality
`!`	$\forall$	Universal Quantifier
`?`	$\exists$	Existential Quantifier
`\`	λ	Lambda Abstraction
`'a`	α	Type Variable Alpha

Figure 2.6: ASCII and Unicode Notations of Common HOL4 Symbols

Writing Definitions

To write a definition in HOL4, the user can use the `Definition` environment. Definitions in HOL4 are made in a functional syntax using equalities. They can contain multiple cases and can be recursive. As an example, we present a HOL4 definition for the factorial of natural numbers in Figure 2.7.

```
Definition fact_def:
  (fact 0       = 1) /\
  (fact (SUC n) = (SUC n) * fact n)
End
```

Figure 2.7: Defintion of the Factorial of a Natural Number in HOL4

When presented with a recursive definition, HOL4 tries to prove the termination of this definition automatically. If this fails, HOL4 requires its user to prove termination manually, usually by providing a measure on the arguments of the definition and by showing that the value of this measure decreases. An example for a termination proof is explained when defining `evaluate` for GCL in Definition B.7 in Appendix B.

HOL4 also allows the definition of algebraic and record data types using the `Datatype` environment. We provide multiple examples of such data types in Appendix B.

Proving Theorems

To prove a theorem in HOL4, the user specifies the theorem to prove in the `Theorem` environment. This term is then loaded into HOL4 as a *proof goal*. By applying *tactics*, the user tells HOL4 how to transform this goal, possibly proving multiple subgoals, until HOL4 can trace the correctness of the remaining goal.

The built-in HOL4 tactics include diverse approaches on proving theorems, including induction, case splits, automatic simplifiers and reasoners, and many domain specific tactics. Tactics can be combined using *tacticals* to provide one single tactic to solve an initial proof goal.

We provide an example proof in Figure 2.8 where we prove that the factorial of $n \geq 2$ is even. The HOL4 proof consists of a tactic which solves the initial proof goal. We explain the details of this proof in Appendix A.

```
Theorem fact_even:
  !n. n >= 2 ==> EVEN (fact n)
Proof
  Induct
  >> rw[fact_def]
  >> Cases_on ‘n >= 2‘
  >| [ALL_TAC, ‘n = 1‘ by gvs[]]
  >> irule (iffRL EVEN_MULT)
  >> gvs[]
QED
```

Figure 2.8: HOL4 Proof: The Factorial of $n \geq 2$ is Even

3. CbC Calculus for Pancake

In this chapter, we describe our CbC calculus for Pancake programs. Our calculus allows developers to refine Hoare specifications into Pancake programs which are formally ensured to have the functionality initially specified.

At first, we define Hoare specifications for Pancake in Section 3.1. In Section 3.2, we prove weakest preconditions of Pancake statements which capture the language's semantics more compactly than the evaluation function. In Section 3.3, we introduce functional and structural contracts for Pancake and their refinement which we complement with refinement rules in Section 3.4.

This chapter contains multiple definitions and theorems which we develop and prove in HOL4, guaranteeing the soundness of our calculus. Within this chapter, we provide our definitions and theorems in both natural language and HOL4 notation but omit the proofs for brevity, as they are developed using HOL4. In Appendix D, we give a complete overview about the theorems and lemmas we used to prove our calculus. In addition to the formalization of our calculus, we provide a case study for evaluation in Chapter 4.

3.1 Hoare Specifications

As Hoare specifications are the initial specification in any CbC process, we need to formalize them based on the programming language's semantics. We introduced the semantics of Pancake and defined Pancake programs and program states in Section 2.2. Based on our definition of program states in Definition 2.12, we continue by defining preconditions and postconditions as predicates on program states and evaluation results:

Definition 3.1 (Preconditions and Postconditions)**.** *A* precondition *is a predicate on program states, i.e., a term of type* α `state` $\rightarrow$ `bool`*.*

A postcondition *is a predicate on tuples of results and program states, i.e., a term of type* α `result option` $\times$ α `state` $\rightarrow$ `bool`*.*

As Definition 2.12 specifies that the clock value is part of the program state, we require a formalization of clockfree conditions. Knowing that conditions are clockfree allows us to use a more simple reasoning about clock values in later proofs. Additionally, it is also not in the interest of users to allow them to reason about clock values, as clock values only exist in the formal semantics but not in the products of the Pancake compiler. We define clock-freeness as follows:

Definition 3.2 (Clockfree Conditions)**.** *A precondition or postcondition is called* clockfree *iff it does not restrict the clock value of the program states.*

$$
\begin{array}{l}
\vdash \texttt{clkfree\_p}\ P \iff \\
\quad \forall s\ k_1\ k_2. \\
\qquad P\ (s\ \texttt{with clock} := k_1) \iff P\ (s\ \texttt{with clock} := k_2) \\
\vdash \texttt{clkfree\_q}\ Q \iff \\
\quad \forall r\ s\ k_1\ k_2. \\
\qquad Q\ (r,s\ \texttt{with clock} := k_1) \iff Q\ (r,s\ \texttt{with clock} := k_2)
\end{array}
$$

Using the common definition of Hoare triples as notations for total correctness in Definition 2.5 as well as the formal definition of Hoare triples for GCL in Definition 2.6, we define Hoare triples for Pancake as follows:

Definition 3.3 (Hoare Triples for Pancake)**.** *A Hoare triple $\{P\}$ prog $\{Q\}$ holds iff both P and Q are clockfree and there is an initial clock value k for any program state s with $P\ s$ such that the program evaluates without an error and without a time out, and the result (r,t) of the evaluation fulfills $Q\ (r,t)$.*

$$
\begin{array}{l}
\vdash \texttt{hoare}\ P\ prog\ Q \iff \\
\quad \texttt{clkfree\_p}\ P \wedge \texttt{clkfree\_q}\ Q \wedge \\
\quad \forall s.\ P\ s \Rightarrow \\
\qquad \exists k.\ (\textbf{let} \\
\qquad\qquad (r,t) = \texttt{evaluate}\ (prog,s\ \texttt{with clock} := k) \\
\qquad\quad \textbf{in} \\
\qquad\qquad r \neq \texttt{SOME Error} \wedge r \neq \texttt{SOME TimeOut} \wedge Q\ (r,t))
\end{array}
$$

In this definition, we exclude two result values: (1) `SOME TimeOut`, as this forces termination according to Definition 2.13, (2) `SOME Error`, as programs executing erroneously are not interesting for the real-world programmer. Additionally, erroneous programs also complicate proving weakest preconditions as done in the following section, as they introduce multiple cases in definitions arising from multiple error sources.

3.2 Weakest Preconditions

The semantics of Pancake is presented by Pohjola et al. as functional big-step semantics [Poh+23], which is convenient for their compiler correctness proofs. For refinement proofs, existing literature usually uses *weakest preconditions* of programs as a starting point [Mor94; KW12]. To follow this path, we define weakest preconditions (WPs) for Pancake programs as follows:

Definition 3.4 (Weakest Preconditions for Pancake Programs). *A program state s fulfills the weakest precondition (WP) of a program prog and a clockfree postcondition Q iff there is an initial clock value k such that the program evaluates from this state without an error and without a time out, and the result* (r,t) *of the evaluation fulfills* $Q\ (r,t)$.

$$
\begin{array}{l}
\vdash \mathit{wp}\ \mathit{prog}\ Q\ s \iff \\
\quad \mathtt{clkfree\_q}\ Q\ \wedge \\
\quad \exists k.\ (\textbf{let} \\
\qquad\qquad (r,t) = \mathtt{evaluate}\ (\mathit{prog},s\ \mathtt{with\ clock}\ := k) \\
\qquad\quad \textbf{in} \\
\qquad\qquad r \neq \mathtt{SOME\ Error} \wedge r \neq \mathtt{SOME\ TimeOut} \wedge Q\ (r,t))
\end{array}
$$

As a first sanity check for this definition, we prove that the WP as defined above is clockfree:

Theorem 3.1 (Clock-Freeness of Weakest Precondition). *The weakest precondition of a Pancake program prog and a postcondition Q is clockfree.*

$$\vdash \mathtt{clkfree\_p}\ (\mathit{wp}\ \mathit{prog}\ Q)$$

Additionally, we ensure that the WP defined above has its two eponymous properties:

Theorem 3.2 (Properties of the Weakest Precondition). *The WP* `wp` *prog Q as defined in Definition 3.4 fulfills that (1) it is a precondition for the program prog and postcondition Q, and (2) it is the weakest of preconditions, i.e., that it is implied by all clockfree preconditions P.*

$$
\begin{array}{l}
\vdash \mathtt{clkfree\_p}\ P \wedge \mathtt{clkfree\_q}\ Q \Rightarrow \\
\quad \mathtt{hoare}\ (\mathit{wp}\ \mathit{prog}\ Q)\ \mathit{prog}\ Q\ \wedge \\
\quad (\mathtt{hoare}\ P\ \mathit{prog}\ Q \iff \forall s.\ P\ s \Rightarrow \mathit{wp}\ \mathit{prog}\ Q\ s)
\end{array}
$$

Using our definition of the WP, we prove the WPs of the different statement types in Pancake. For brevity, we only demonstrate the WP of the `Skip` and `Seq` $p_1\ p_2$ statements in this section, each with a proof sketch to show the general approach on these proofs and the included construction of clock values. The remaining WPs can be found in Appendix D.

Theorem 3.3 (Weakest Precondition of `Skip`). *The WP of* `Skip` *and the postcondition Q is exactly fulfilled by the states s which fulfill Q* (NONE,s)*, i.e., all states allowed with normal termination in Q.*

$$\vdash \mathtt{clkfree\_q}\ Q \Rightarrow (\mathit{wp}\ \mathtt{Skip}\ Q\ s \iff Q\ (\mathtt{NONE},s))$$

Proof Sketch. By applying Definitions 3.2 and 3.4 and the definition of the evaluation function (see Appendix C), we have to show that

$$\frac{\forall r\ s\ k_1\ k_2.\ \ Q\ (r,s\ \mathtt{with\ clock}\ := k_1) \iff Q\ (r,s\ \mathtt{with\ clock}\ := k_2)}{Q\ (\mathtt{NONE},s) \iff \exists k.\ \ Q\ (\mathtt{NONE},s\ \mathtt{with\ clock}\ := k)}$$

holds. For the forward implication, let $k = s$.clock. For the reverse implication, use the assumption instantiated with (r,s,k_1,k_2) = (NONE,s,k,s.clock). ◇

Theorem 3.4 (Weakest Precondition of `Seq` p_1 p_2)**.** *The WP of* `Seq` p_1 p_2 *and the postcondition Q is exactly fulfilled by the states s which fulfill (1) the WP of p_1 and the postcondition* λ `(`r`,`t`).` r `= NONE` $\wedge$ `wp` p_2 Q t *, or (2) the WP of p_1 and the postcondition* λ `(`r`,`t`).` $r \neq$ `NONE` $\wedge$ Q `(`r`,`t`)` *.*

$$
\begin{array}{l}
\vdash \texttt{clkfree\_q}\ Q \Rightarrow \\
\quad (\texttt{wp}\ (\texttt{Seq}\ p_1\ p_2)\ Q\ s \iff \\
\quad\ \ \texttt{wp}\ p_1\ (\lambda\,(r,t).\ r\ \texttt{= NONE} \wedge \texttt{wp}\ p_2\ Q\ t)\ s \vee \\
\quad\ \ \texttt{wp}\ p_1\ (\lambda\,(r,t).\ r \neq \texttt{NONE} \wedge Q\ (r,t))\ s)
\end{array}
$$

Proof Sketch. After applying Definitions 3.2 and 3.4 and the definition of the evaluation function (see Appendix C), we continue as follows:

For the forward implication, we distinguish two cases: (1) If the evaluation of p_1 results in `NONE`, we have to show that there is an initial clock value k' for the successive execution of p_1 and p_2 as given by the nested WPs in the consequent. This nested execution has to terminate in a result fulfilling Q `(`r`,`t`)`. The necessary clock value can be constructed using Theorems 2.6 and 2.7 from the initial clock values of the single executions of p_1 and p_2. The result fulfills Q `(`r`,`t`)` by construction as required by the antecedent of the implication. (2) If the evaluation of p_1 does not result in `NONE`, we have to show that there is an initial clock value k' for the execution of p_1 as given by the WP in the consequent. This execution has to terminate in a result fulfilling Q `(`r`,`t`)`. The necessary clock value is exactly the clock value from the antecedent, which also guarantees Q `(`r`,`t`)`.

For the reverse implication, we prove the implication for both disjunctive terms in the antecedent: (1) Given the successive execution of p_1 and p_2 as given with the nested WPs in the antecedent, we have to show that there is an initial clock value for the execution of `Seq` p_1 p_2, and that this execution terminates in a result fulfilling Q `(`r`,`t`)`. After applying Theorem 2.7, this clock value can be constructed as the sum of the minimal clock values for p_1 and p_2, and the result fulfills Q `(`r`,`t`)` by construction. (2) Given the execution of p_1 to a result not equal to `NONE` as given with the WP in the antecedent, we have to show that there is an initial clock value for the execution of p_1, and that this execution terminates in a result fulfilling Q `(`r`,`t`)`. The necessary clock value is exactly the clock value from the antecedent, which also guarantees Q `(`r`,`t`)`. ◊

In general, we use these WPs to prove refinement rules for Pancake in Section 3.4, with the exception of the rule for while loops. As the weakest precondition in Section 3.2 is a specification of total correctness, it also specifies the termination of a program. Therefore, it is not possible to find an expressive notation of a while loop's WP as we prove in the following theorem:

Theorem 3.5 (Weakest Precondition of While Loop)**.** *There are Pancake while loops for which it is not decidable whether a state fulfills the WP of the loop.*

Proof. Let *tmProg* be the Pancake program that simulates a Turing machine, structured as a while loop doing one step of the Turing machine per iteration. If it was decidable whether a state fulfilled the WP of the main loop in *tmProg*, it would decide whether the simulated Turing machine would halt, as the WP is a specification of total correctness, and therefore deciding the halting problem. □

3.3 Contracts and Refinement

Following the path we sketched for GCL in Chapter 2, we continue by specifying a language of contracts which we use in our refinement calculus. According to our informal definitions in Section 2.1.1, we define the language of contracts for our subset of Pancake as follows:

Definition 3.5 (Pancake Contracts)**.** *The contracts to be used for refinement-based programming in Pancake are terms of the datatype shown below. The types* α `prog` *and* α `exp` *denote programs and expressions (see Figure 2.2),* α `state` *denotes program states as defined in Figure 2.4, and* α `result` *denotes possible evaluation results listed in Appendix C.*

```
α Contract =
    HoareC (α state → bool) (α result option × α state → bool)
  | DecC mlstring shape (α exp) (α Contract)
  | SeqC (α Contract) (α Contract)
  | IfC (α exp) (α Contract) (α Contract)
  | WhileC (α exp) (α state → bool) (α state → num) (α Contract)
  | PanC (α prog)
  | DCC
```

As the Pancake semantics allows diverging programs, other than the semantics of GCL in Section 2.1, we have to add a *variant* to allow us to force the termination of loops, similiarly to the GCL semantics discussed in Appendix B. We define a variant of a loop as follows:

Definition 3.6 (Loop Variant)**.** *A loop variant v for the loop body p is a clockfree term of type* α `state` → `num` *whose value decreases after evaluation of p given that the invariant i holds.*

```
⊢ is_variant i v p ⟺
  (∀s. i s ⇒ v (SND (evaluate (p,s))) < v s) ∧
  ∀s k₁ k₂.
    v (s with clock := k₁) = v (s with clock := k₂)
```

As done for GCL in Definition 2.7, we continue by defining the satisfication relation for the contracts defined above:

Definition 3.7 (Satisfication of Pancake Contracts)**.** *The satisfication relation* **sat** ⊆ *prog* × *Contract for Pancake contracts is defined by the following equalities. For all remaining cases, the reader may assume non-satisfication.*

```
⊢ sat (HoareC P Q) prog ⟺ hoare P prog Q
⊢ sat (DecC nl sl el c) (Dec nr sr er p) ⟺
  nl = nr ∧ sl = sr ∧ el = er ∧ sat c p
⊢ sat (SeqC c₁ c₂) (Seq p₁ p₂) ⟺ sat c₁ p₁ ∧ sat c₂ p₂
⊢ sat (IfC l c₁ c₂) (If r p₁ p₂) ⟺
  l = r ∧ sat c₁ p₁ ∧ sat c₂ p₂
⊢ sat (WhileC l i v c) (While r p) ⟺
  l = r ∧ sat c p ∧ is_variant i v p
⊢ sat (PanC l) r ⟺ l = r
⊢ sat DCC v₀ ⟺ T
```

In these equalities, `hoare` P $prog$ Q *is the Hoare triple relation as defined in Definition 3.3 and* `is_variant` i v p *is the requirement that* v *is a variant for the program* p *as defined in Definition 3.6.*

In Section 2.1.2, we use the satisfication relation to define a refinement relation between contracts. Similarly, we define the refinement relation for Pancake contracts:

Definition 3.8 (Refinement of Pancake Contracts)**.** *Let* $\mathsf{sat} \subseteq \mathsf{prog} \times \mathsf{Contract}$ *be the satisfication relation between programs and contracts, i.e.,* p sat c *iff* p *satisfies* c*.*

A contract c_1 *can be refined into a contract* c_2*, denoted by* `refine` c_1 c_2*, iff the satisfication of* c_2 *implies the satisfication of* c_1*.*

$$\vdash \texttt{refine}\ c_1\ c_2 \iff \forall prog.\ \texttt{sat}\ c_2\ prog \Rightarrow \texttt{sat}\ c_1\ prog$$

As proven in Theorem 2.1, the refinement relation between contracts is both reflexive and transitive without knowledge of the definition of the satisfication relation. Therefore, these properties also hold for the refinement of Pancake contracts. As for the GCL contracts, the composition of Pancake contracts is also monotonic with respect to the refinement relation:

Theorem 3.6 (Monotonicity of Refinement)**.** *The composition of contracts is monotonic with respect to the refinement relation.*

$$\begin{aligned}
&\vdash \texttt{refine}\ A\ B \Rightarrow \texttt{refine (DecC}\ v\ sh\ exp\ A\texttt{) (DecC}\ v\ sh\ exp\ B\texttt{)}\\
&\vdash \texttt{refine}\ A\ B \Rightarrow\\
&\quad \texttt{refine (SeqC}\ A\ C\texttt{) (SeqC}\ B\ C\texttt{)} \land\\
&\quad \texttt{refine (SeqC}\ C\ A\texttt{) (SeqC}\ C\ B\texttt{)}\\
&\vdash \texttt{refine}\ A\ B \Rightarrow\\
&\quad \texttt{refine (IfC}\ e\ A\ C\texttt{) (IfC}\ e\ B\ C\texttt{)} \land\\
&\quad \texttt{refine (IfC}\ e\ C\ A\texttt{) (IfC}\ e\ C\ B\texttt{)}\\
&\vdash \texttt{refine}\ A\ B \Rightarrow \texttt{refine (WhileC}\ e\ i\ v\ A\texttt{) (WhileC}\ e\ i\ v\ B\texttt{)}
\end{aligned}$$

3.4 Refinement Rules

Using the definition of refinement from the preceding section, we continue with proving refinement rules for Pancake programs as discussed for GCL in Section 2.1.3.

In this section, we discuss two types of refinement rules: in Sections 3.4.1 to 3.4.4, we prove *top-down* refinement rules, i.e., rules that introduce new contracts starting from `HoareC` specifications. For this part, we present the refinement rules used in the case study in Chapter 4, which are the refinement rules for `Skip`, variable declarations and assignments, program sequencing, conditionals, while loops, and return statements. Of these, we provide proof sketches for the rules for `Skip`, program sequencing, and while loops. The remaining refinement rules are listed in Appendix D.

In Section 3.4.5, we additionally discuss *bottom-up* refinement rules, i.e. rules from composite contracts like `SeqC` and `WhileC` to `PanC` contracts. These refinement rules finalize the refinement of sub-programs and are usually trivial to apply with an exception for the rule dealing with `WhileC` which ensures the termination of the loop.

We provide an overview of all refinement rules in Appendix D.

3.4.1 Refinement Rule: `Skip`

Inspecting the WP of `Skip` proven in Theorem 3.3 as well as the definition of refinement in Definition 3.8, we obtain three side conditions for a refinement rule for `Skip`: (1) the clock-freeness of the precondition, (2) the clock-freeness of the postcondition, and (3) the precondition implying the postcondition, fixed to normal termination. Therefore, we prove:

Theorem 3.7 (Refinement Rule: `Skip`).

$\vdash$ *clkfree_p* P $\wedge$ *clkfree_q* Q $\wedge$ $(\forall s.\ P\ s \Rightarrow Q$ *(NONE,*s*))* $\Rightarrow$
refine (HoareC P Q*) (PanC Skip)*

Proof Sketch. After applying Definitions 3.7 and 3.8, it remains to be shown that the side condition

`clkfree_p` P $\wedge$ `clkfree_q` Q $\wedge$ $\forall s.\ P\ s \Rightarrow Q$ `(NONE,`s`)`

implies `hoare` P `Skip` Q.

Using the right-to-left implication of the second conjunct in Theorem 3.2, the term above has to imply

$(\forall s.\ P\ s \Rightarrow$ `wp Skip` $Q\ s)$ $\wedge$ `clkfree_p` P $\wedge$ `clkfree_q` Q.

The second and third conjunct is directly implied by the side condition, the first cunjunct is solved by applying Theorem 3.3. ◊

3.4.2 Refinement Rule: `Seq` p_1 p_2

For program sequencing, we observe two cases in the WP proven in Theorem 3.4: if p_1 evaluates successfully, the resulting state has to fulfill the WP of p_2. Otherwise, this resulting state has to fulfill the postcondition Q, as p_2 is not executed.

Due to the multiple cases in the WP, we are also able to provide multiple refinement rules for program sequencing, each targeting one or both of the cases above. For this section, we focus on the refinement rule which allows a termination in both subprograms p_1 and p_2. The refinement rules targeting the single cases are shown in Appendix D.

As for `Skip`, we see in Definition 3.8, that the clock-freeness of both precondition and postcondition is a necessary side condition for the sequence rule. Additionally, the intermediate condition M reached after evaluating p_1 needs to be clockfree, as it possibly is a precondition for p_2. For the subcontracts in `SeqC` c_1 c_2, we choose

`HoareC` P $(\lambda(r,t).$ **if** $r \neq$ `NONE` **then** $Q\ (r,t)$ **else** $M\ t)$

and `HoareC` M Q, as this resembles the cases in the WP and allows a termination in both subprograms. Therefore, we prove

Theorem 3.8 (Refinement Rule: `Seq` p_1 p_2).

$\vdash$ *clkfree_p* P $\wedge$ *clkfree_q* Q $\wedge$ *clkfree_p* M $\Rightarrow$
refine (HoareC P Q*)*
  (SeqC
    (HoareC P
      $(\lambda(r,t).$ **if** $r \neq$ *NONE* **then** $Q\ (r,t)$ **else** $M\ t))$
    (HoareC M Q*))*

Proof Sketch. After applying Definitions 3.7 and 3.8, it remains to be shown that the side conditions and the right-hand side of the WP, i.e.,

$$\begin{aligned}&\texttt{clkfree\_p}\ P \wedge \texttt{clkfree\_q}\ Q \wedge \texttt{clkfree\_p}\ M \wedge\\ &\texttt{hoare}\ P\ p_1\ (\lambda(r,t).\ \textbf{if}\ r \neq \texttt{NONE}\ \textbf{then}\ Q\ (r,t)\ \textbf{else}\ M\ t) \wedge\\ &\texttt{hoare}\ M\ p_2\ Q\end{aligned}$$

imply $\texttt{hoare}\ P\ (\texttt{Seq}\ p_1\ p_2)\ Q$.

Using the right-to-left implication of the second conjunct in Theorem 3.2 on both assumptions and the goal, and after solving both clock-freeness subgoals $\texttt{clkfree\_p}\ P$ and $\texttt{clkfree\_q}\ Q$ with the corresponding side conditions, it remains to be shown that the postcondition for p_1 is clockfree, i.e.,

$$\texttt{clkfree\_q}\ (\lambda(r,t).\ \textbf{if}\ r \neq \texttt{NONE}\ \textbf{then}\ Q\ (r,t)\ \textbf{else}\ M\ t)$$

and that the assumptions

$$\begin{aligned}&\forall s.\ P\ s \Rightarrow \texttt{wp}\ p_1\ (\lambda(r,t).\ \textbf{if}\ r \neq \texttt{NONE}\ \textbf{then}\ Q\ (r,t)\ \textbf{else}\ M\ t)\ s\\ &\forall s.\ M\ s \Rightarrow \texttt{wp}\ p_2\ Q\ s\end{aligned}$$

imply $\forall s.\ P\ s \Rightarrow \texttt{wp}\ (\texttt{Seq}\ p_1\ p_2)\ Q\ s$.

In the remainder of this proof, we use a set of monotonicity theorems for predicate constructions and WPs. These are listed in Appendix D and only referenced here.

We solve the clock-freeness subgoal using Theorem D.12 and continue with the WP subgoal by applying Theorem D.22 to the first assumption. This results in a new assumption stack:

$$\begin{aligned}&\forall s.\ P\ s \Rightarrow\\ &\quad \texttt{wp}\ p_1\ (\lambda(r,t).\ r \neq \texttt{NONE} \wedge Q\ (r,t))\ s \vee\\ &\quad \texttt{wp}\ p_1\ (\lambda(r,t).\ r = \texttt{NONE} \wedge M\ t)\ s\\ &\forall s.\ M\ s \Rightarrow \texttt{wp}\ p_2\ Q\ s\end{aligned}$$

Using the theorem Theorem D.13 on the second assumption, we obtain

$$\begin{aligned}&\forall s.\ P\ s \Rightarrow\\ &\quad \texttt{wp}\ p_1\ (\lambda(r,t).\ r \neq \texttt{NONE} \wedge Q\ (r,t))\ s \vee\\ &\quad \texttt{wp}\ p_1\ (\lambda(r,t).\ r = \texttt{NONE} \wedge M\ t)\ s\\ &\forall s.\ (\lambda(r,t).\ r = \texttt{NONE} \wedge M\ t)\ s \Rightarrow\\ &\quad (\lambda(r,t).\ r = \texttt{NONE} \wedge \texttt{wp}\ p_2\ Q\ t)\ s.\end{aligned}$$

Using Theorem D.21 on the second assumption yields the new assumptions

$$\begin{aligned}&\forall s.\ P\ s \Rightarrow\\ &\quad \texttt{wp}\ p_1\ (\lambda(r,t).\ r \neq \texttt{NONE} \wedge Q\ (r,t))\ s \vee\\ &\quad \texttt{wp}\ p_1\ (\lambda(r,t).\ r = \texttt{NONE} \wedge M\ t)\ s\\ &\forall s.\ (\lambda(r,t).\ r = \texttt{NONE} \wedge M\ t)\ s \Rightarrow\\ &\quad (\lambda(r,t).\ r = \texttt{NONE} \wedge \texttt{wp}\ p_2\ Q\ t)\ s,\end{aligned}$$

having to prove the clock-freeness of both predicates

$$\begin{aligned}&\lambda(r,t).\ r = \texttt{NONE} \wedge M\ t\\ &\lambda(r,t).\ r = \texttt{NONE} \wedge \texttt{wp}\ p_2\ Q\ t.\end{aligned}$$

The clock-freeness can be proved using Theorems D.9 and D.18. The remaining goal is solved by applying the WP proven in Theorem 3.4. ◊

3.4.3 Refinement Rule: `While` e p

As shown in Theorem 3.5, there is no notation for the WP of a while loop, such that it is decidable whether a state fulfills the WP. Therefore, we cannot use the proof pattern presented for `Skip` and `Seq` p_1 p_2 above to prove a refinement rule for the while loop.

To allow the refinement of while loops, we follow a similar path as done for GCL in Section 2.1.3 and Appendix B: we introduce a refinement rule that uses an invariant to demonstrate correct functionality as well as a variant to demonstrate termination.

For the invariant, we set similar side conditions as for GCL, i.e., that (1) the precondition implies the invariant, that (2) the invariant and the negation of the guard implies the postcondition fixed to normal termination, and (3) the loop body re-establishes the invariant when started from the invariant.

As Pancake has different properties than GCL, we have to make adaptions to these properties: Pancake supports the abnormal termination of loops, i.e., the termination of a loop due to exceptions, *break*, and *return*. This requires us to tweak (3), such that the invariant only needs to be re-established when there was no abnormal termination in this iteration. It also requires us to amend to (2), linking the abnormal termination of the loop body to the abnormal termination of the whole loop. This is done by adding *sub-postconditions*, implying special cases of the postcondition. In addition to (1), we have to take into account that Pancake expressions do not necessarily evaluate in any state, adding the requirement that the invariant implies that the guard can be evaluated.

For the variant, we do not introduce requirements in the while refinement rule, as the requirement to a variant is already introduced in the satisfication relation defined in Definition 3.7. As `is_variant` i v p is defined on Pancake programs p, this only can be resolved within the refinement of a `WhileC` to a `PanC` discussed in Section 3.4.5.

We continue by introducing two abbreviations used in the while refinement rule:

Definition 3.9 (Precondition and Postcondition for While Loop Bodies)**.** *We define the following shorthand notations where* `evaluates_to_true` e s *is fulfilled iff the expression* e *evaluates to* true *from state* s *and* QB*,* QR*, and* QE *are the* sub-postconditions *for termination due to* break*,* return *and exceptions.*

```
⊢ while_body_pre i e = (λs. i s ∧ evaluates_to_true e s)
⊢ while_body_post i QB QR QE =
   (λ(r,t).
         case r of
           NONE ⇒ i t
         | SOME Error ⇒ i t
         | SOME TimeOut ⇒ i t
         | SOME Break ⇒ QB t
         | SOME Continue ⇒ i t
         | SOME (Return v) ⇒ QR (t,v)
         | SOME (Exception eid e) ⇒ QE (t,eid,e))
```

Formalizing the requirements stated above, we introduce the following refinement rule, providing only a rough proof sketch:

Theorem 3.9 (Refinement Rule: `While` e p)**.** *Let the predicates* `evaluates_to_word` *and* `evaluates_to_false` *be fulfilled iff an expression evaluates to a word or to* false *in a given state. In this case, it holds that*

```
clkfree_p P ∧ clkfree_q Q ∧ clkfree_p i ∧ (∀s. P s ⇒ i s) ∧
(∀s. i s ⇒ evaluates_to_word e s) ∧
(∀s. i s ∧ evaluates_to_false e s ⇒ Q (NONE,s)) ∧
(∀t. QB t ⇒ Q (NONE,t)) ∧
(∀t v. QR (t,v) ⇒ Q (SOME (Return v),t)) ∧
(∀t eid v. QE (t,eid,v) ⇒ Q (SOME (Exception eid v),t)) ⇒
refine (HoareC P Q)
  (WhileC e i v
     (HoareC (while_body_pre i e)
        (while_body_post i QB QR QE)))
```

Proof Sketch. After applying Definitions 3.3, 3.7, and 3.8, and the abbreviations above, we have to prove that

```
∀s. P s ⇒
    ∃k. (λ(r,t).
              r ≠ SOME Error ∧ r ≠ SOME TimeOut ∧ Q (r,t))
           (evaluate (While e p,s with clock := k))
```

holds as a consequence of the given assumptions in the goal's antecedent. In the remainder of this proof, we will refer to these assumptions by their index in the goal above.

We continue by replacing P s with i s in the antecedent, justified by the fourth assumption, and commence a inductive proof using the variant v as a measure to induct on. This allows us to input the definition of `evaluate` which is shown in Figure 2.5 in the relevant excerpts.

To simplify the goal, we use the fifth assumption as well as Theorem 2.5 to reduce the case expression to its second case.

The remaining goal now contains two branches, one per evaluation result of the guard. To solve the terminating case, i.e., the guard evaluating to false, we use the sixth assumption and specify the required clock value as s.clock. We continue with the case in which the guard does not cause termination.

We now may use the assumption obtained by the right-hand-side of the refinement rule, as `while_body_pre` i e s is now fulfilled by excluding the guard evaluating to false, yielding `while_body_post` i QB QR QE s as a new assumption.

We distinguish two cases on how to construct the required clock value: (1) The loop continuing after the current iteration, i.e., the body evaluating to `NONE` or `SOME Continue`, or causing an `Error` or a `TimeOut`, and (2) the loop abnormally terminating, i.e., the body evaluating to any other result.

For (1), we use the clock-freeness of i from the third assumption to show that the invariant also holds for the clock value k necessary to evaluate the loop body. Applying the definition and the clock-freeness of the variant, guaranteed by `is_variant` in the satisfication relation, we obtain v t $<$ v s where t is the program state after evaluating the loop body.

This allows us to use the induction hypothesis introduced earlier, yielding that there is a clock value such that the recursive evaluation of the loop following after the body will terminate, starting with an initial clock value k'.

By applying Theorem 2.7 to both the loop body and the recursive evaluation of the loop, yielding minimal clock values k'' and $k'^{3\prime}$, we use Theorem 2.6 to show that the loop body will also terminate from the clock value $k'' + k'^{3\prime}$. By providing $k'' + k'^{3\prime} + 1$ as clock value for the existential quantifier in the goal, we solve this first case.

For (2), it suffices to provide $k + 1$ as clock value for the existential quantifier in the goal, solving the second case with respect to the seventh, eighth, and ninth assumption. $\Diamond$

3.4.4 Other Top-Down Refinement Rules

The remaining top-down refinement rules we use in Chapter 4 are presented in the theorems below. Their proofs follow the structure of the proof for Theorem 3.8 and are therefore omitted for brevity.

Refinement Rules: `Dec` *v sh src prog*

For the variable declaration `Dec`, we provide two refinement rules: both require the newly declared variable to use a fresh variable name, which is without loss of generality but simplifies their application. The first rule targets variable declarations from expressions with known value, usually constants, while the second rule targets variable declarations from memory loads.

In the theorems below, the predicates `varfree_p` and `varfree_q` ensure that the precondition and postcondition do not contain the variable to be declared, as the assumption ¬`MEM` *v* (`var_exp` *ad*) ensures that the expression for the memory address does not contain the variable as well. The predicates `evaluates_to` and `evaluates_shape` ensure that an expression evaluates to the given value or to a value of the given shape. The predicates `var_eq_val` and `var_eq_mem` ensure that the provided variable has the given value or the value at the given point in memory.

Theorem 3.10 (Refinement Rule: `Dec` *v* `One` *src prog* with known value of *src*)**.**

```
⊢ clkfree_p P ∧ clkfree_q Q ∧ varfree_p v P ∧ varfree_q v Q ∧
  (∀s. P s ⇒ evaluates_to src val s) ⇒
  refine (HoareC P Q)
    (DecC v sh src
       (HoareC (λs. P s ∧ var_eq_val Local v val s) Q))
```

Theorem 3.11 (Refinement Rule: `Dec` *v sh* (`Load` *sh ad*) *prog*)**.**

```
⊢ clkfree_p P ∧ clkfree_q Q ∧ varfree_p v P ∧ varfree_q v Q ∧
  (∀s. P s ⇒ evaluates_shape (Load sh ad) sh s) ∧
  ¬MEM v (var_exp ad) ⇒
  refine (HoareC P Q)
    (DecC v sh (Load sh ad)
       (HoareC (λs. P s ∧ var_eq_mem Local v ad sh s) Q))
```

Refinement Rule: `Assign` k v src

For variable assignments to local and global variables, i.e., both k `= Local` and k `= Global`, we provide one refinement rule.

In the theorem below, the predicate `valid_value` ensures that the value to be assigned is of the same shape as the existing value of the variable. The predicate operator `subst` is defined as follows:

Definition 3.10 (Variable Substitution)**.** *A variable is substituted in a predicate to the value of an expression, if this expression evaluates and a state updated with this value fulfills the predicate.*

```
⊢ subst k v e P s ⟺
    ∃value.
      eval s e = SOME value ∧
      case k of
        Local ⇒ P (s with locals := s.locals |+ (v,value))
      | Global ⇒
        P (s with globals := s.globals |+ (v,value))
```

Theorem 3.12 (Refinement Rule: `Assign` k v src)**.**

```
⊢ clkfree_p P ∧ clkfree_q Q ∧
    (∀s. P s ⇒
         valid_value k v src s ∧
         subst k v src (λs. Q (NONE,s)) s) ⇒
    refine (HoareC P Q) (PanC (Assign k v src))
```

Refinement Rule: `If` e p_1 p_2

For the conditional, we provide one refinement rule.

In the theorem below, the predicates `evaluates_to_word`, `evaluates_to_true`, and `evaluates_to_false` ensure that an expression can be interpreted as a boolean value, evaluates to *true*, or evaluates to *false*.

Theorem 3.13 (Refinement Rule: `If` e p_1 p_2)**.**

```
⊢ clkfree_p P ∧ clkfree_q Q ∧
    (∀s. P s ⇒ evaluates_to_word e s) ⇒
    refine (HoareC P Q)
      (IfC e (HoareC (λs. P s ∧ evaluates_to_true e s) Q)
         (HoareC (λs. P s ∧ evaluates_to_false e s) Q))
```

Refinement Rule: `Return` r

For the return statement, we provide one refinement rule.

In the theorem below, the predicate `evaluates_to` e val s ensures that the provided expression evaluates to a value val. `size_of_shape (shape_of` val`)` ≤ 32 requires this value to have a maximum size of 32 words. The operator `empty_locals` empties the local values in a program state.

Theorem 3.14 (Refinement Rule: `Return` r).

```
⊢ clkfree_p P ∧ clkfree_q Q ∧
  (∀s. P s ⇒
      ∃val.
        evaluates_to e val s ∧
        size_of_shape (shape_of val) ≤ 32 ∧
        Q (SOME (Return val),empty_locals s)) ⇒
  refine (HoareC P Q) (PanC (Return e))
```

3.4.5 Bottom-Up Refinement Rules

The refinement rules presented in the preceding sections each introduce new contracts, either composite contracts like `SeqC` and `WhileC` or program contracts `PanC`. After applying these refinement rules, the developer ultimately reaches the refinement into a contract which does not allow any further refinement but is also not a program contract `PanC`. To achieve a program contract as the final result of our refinement process, we are required to add bottom-up rules that allow the refinement of composite contracts to program contracts.

In general, these rules are trivial as the following example shows:

Theorem 3.15 (Bottom-Up Refinement Rule: `Seq` p_1 p_2).

```
⊢ refine (SeqC (PanC l) (PanC r)) (PanC (Seq l r))
```

Proof Sketch. Apply Definitions 3.7 and 3.8. ◊

For `WhileC`, the satisfication relation requires us to prove the variant within this refinement as proven in the following theorem:

Theorem 3.16 (Bottom-Up Refinement Rule: `While` e p).

```
⊢ is_variant i v p ⇒
  refine (WhileC e i v (PanC p)) (PanC (While e p))
```

Proof Sketch. Apply Definitions 3.7 and 3.8. ◊

4. Case Study: Linear Search

In this chapter, we demonstrate the feasibility of our CbC calculus for Pancake by providing a refinement proof for an implementation of linear search, using the refinement rules we introduce in Section 3.4. The resulting Pancake program of this chapter is structured as the program shown in Figure 2.3, while making two adaptions: (1) we model the function inputs as constants and (2) we require that the sought-after item appears in the provided array.

The remainder of this chapter is structured as follows: in Section 4.1, we develop a formal specification for linear search, which we refine into a Pancake program in Section 4.2. In Section 4.3, we discuss our current progress towards automatic step-wise refinement proofs in HOL4 as well as challenges arising with larger case studies.

As in the preceding chapters, we omit the detailled proofs, as they are developed in HOL4.

4.1 Specification

To formalize the functionality of an implementation of linear search in Pancake, we need to inspect both the properties of the algorithm itself as well as the properties of the language's semantics which require us to introduce additional conditions.

Properties of the Algorithm

Beginning with the properties of the algorithm, we specify the functionality of any search algorithm: Our implementation of linear search shall receive an array, i.e., a base address and an array length, and a word to search for in this array. We require that this word appears in the array. The implementation shall return a memory address within the array, at which there is an occurrence of the provided word. To formalize these properties, we intoduce two abbreviations, where $(<_+)$ and $(\leq_+)$ are unsigned comparison operators:

Definition 4.1 (Appearance of Words in Arrays).

$$\vdash \mathit{appears}\ x\ b\ l = (\lambda s.\ \exists \mathit{addr}.\ b \leq_+ \mathit{addr} \land \mathit{addr} <_+ b + l \land s.\mathsf{memory}\ \mathit{addr} = x)$$

Definition 4.2 (Memory Equality)**.**

$\vdash$ `mem_eq` x $addr$ = (λs. s.`memory` $addr$ = x)

Using these definitions, we can set the following terms as conjuncts within our precondition and postcondition:

```
$P_1$ = ($\lambda s$. appears (Word $x$) $b$ $l$ $s$)
$Q_1$ =
($\lambda$ ($r$,$t$).
     $\exists ret$.
       $r$ = SOME (Return (ValWord $ret$)) $\wedge$ $b \leq_+ ret$ $\wedge$
       $ret <_+ b + l$ $\wedge$ mem_eq (Word $x$) $ret$ $t$)
```

As an example for relevant side conditions of searching, we additionally introduce the requirement that the search algorithm leaves the memory unchanged, yielding the following additional conjuncts:

```
$P_2$ = ($\lambda s$. $s$.memory = $the\_mem$)
$Q_2$ = ($\lambda$ ($r$,$t$). $t$.memory = $the\_mem$)
```

Properties of the Language's Semantics

As shown in Figure 2.4, the Pancake semantics models memory as a function from α `word` to α `word_lab`, where α `word_lab` is an algebraic data type which used to contain a word or a *function label* in earlier versions of Pancake. Since function labels were removed in April 2025[1], α `word_lab` may only contain words, effectively making the memory model a function from α `word` to α `word`.

```
⊢ eval $s$ (Load $shape$ $addr$) =
  case eval $s$ $addr$ of
    NONE ⇒ NONE
  | SOME (ValWord $w$) ⇒ mem_load $shape$ $w$ $s$.memaddrs $s$.memory
  | SOME (Struct $v_5$) ⇒ NONE
⊢ mem_load $sh$ $addr$ $dm$ $m$ =
  case $sh$ of
    One ⇒ if $addr \in dm$ then SOME (Val ($m$ $addr$)) else NONE
  | Comb $shapes$ ⇒
    case mem_loads $shapes$ $addr$ $dm$ $m$ of
      NONE ⇒ NONE
    | SOME $vs$ ⇒ SOME (Struct $vs$)
```

Figure 4.1: Pancake Semantics: Evaluation of Memory Loads

Inspecting the semantic's definition of memory loads shown in Figure 4.1, we deduct another requirement to be added: `mem_load One`, which we use to fetch items from the array, checks if the provided address is member of the domain s.memaddrs. Thus, we have to ensure that this is the case for all items in our array.

Additionally, as we do arithmetic on fixed-width words, we have to ensure that our array specification consisting of base address and array length is sound, i.e., that there is no overflow between base address and the last item of the array.

[1]Commit `5712647` of the Git repository at https://github.com/CakeML/cakeml

As both properties characterize an array as an accessible and contiguous part of the memory, we summarize them in the following abbreviation:

Definition 4.3 (Array in Memory)**.**

```
⊢ array_in_mem b l =
  (λ s.
       b <+ b + l ∧
       ∀ addr. b ≤+ addr ∧ addr <+ b + l ⇒ s.memaddrs addr)
```

Using this as a last conjunct for our precondition, we define the specification for the remainder of this chapter as

```
HoareC
  (λ s.
       appears (Word x) b l s ∧ s.memory = the_mem ∧
       array_in_mem b l s)
  (λ (r,t).
       t.memory = the_mem ∧
       ∃ ret.
         r = SOME (Return (ValWord ret)) ∧ b ≤+ ret ∧
         ret <+ b + l ∧ mem_eq (Word x) ret t),
```

additionally initializing α `word` as `word64` to simplify some later proof steps.

4.2 Refinement Proof

We begin the refinement proof in top-down direction by closely following the example shown in Figure 2.3. After finishing the top-down direction in Step 8, we continue with the bottom-up direction, mostly using transitivity (Theorem 2.1), monotonicity (Theorem 3.6), and bottom-up refinement rules.

To visualize the process, we present the refinement steps as a graphs in Figures 4.2 to 4.7. An edge between two contract edges c_1 and c_2 represents `refine` c_1 c_2.

Step 1: Initialize Counter

We decide to begin with a variable declaration, initializing a counter variable as zero. This can be done by applying the declaration refinement rule for known values shown in Theorem 3.10. This refinement step is shown in Figure 4.2.

To fulfill the proof goal

```
refine (HoareC P Q)
  (DecC v One (Const 0w)
     (HoareC (λ s. P s ∧ var_eq_val Local v (ValWord 0w) s)
        Q)),
```

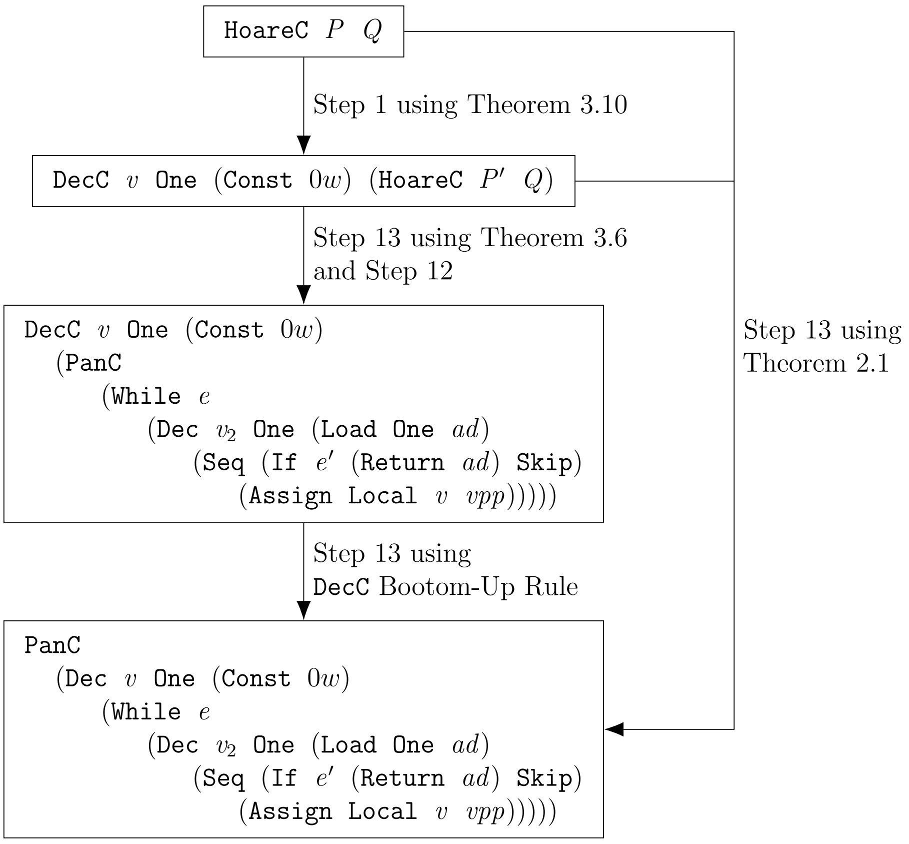
we have to prove the side conditions

```
clkfree_p P ∧ clkfree_q Q ∧ varfree_p v P ∧ varfree_q v Q ∧
∀ s. P s ⇒ evaluates_to (Const 0w) (ValWord 0w) s,
```

i.e., the clock-freeness and variable-freeness of precondition and postcondition, and that the precondition implies the evaluation of `Const 0w` to `ValWord 0w`.

These goals can be solved by applying Definitions 3.2 and 4.1 to 4.3, as well as the definitions of `varfree_p` and `varfree_q`. For brevity, we will refer to the precondition of the resulting subcontract as

```
P′ = (λ s. P s ∧ var_eq_val Local v (ValWord 0w) s).
```



```
HoareC P Q
```

```
DecC v One (Const 0w) (HoareC P′ Q)
```

```
DecC v One (Const 0w)
  (PanC
     (While e
        (Dec v₂ One (Load One ad)
           (Seq (If e′ (Return ad) Skip)
              (Assign Local v vpp)))))
```

```
PanC
  (Dec v One (Const 0w)
     (While e
        (Dec v₂ One (Load One ad)
           (Seq (If e′ (Return ad) Skip)
              (Assign Local v vpp)))))
```

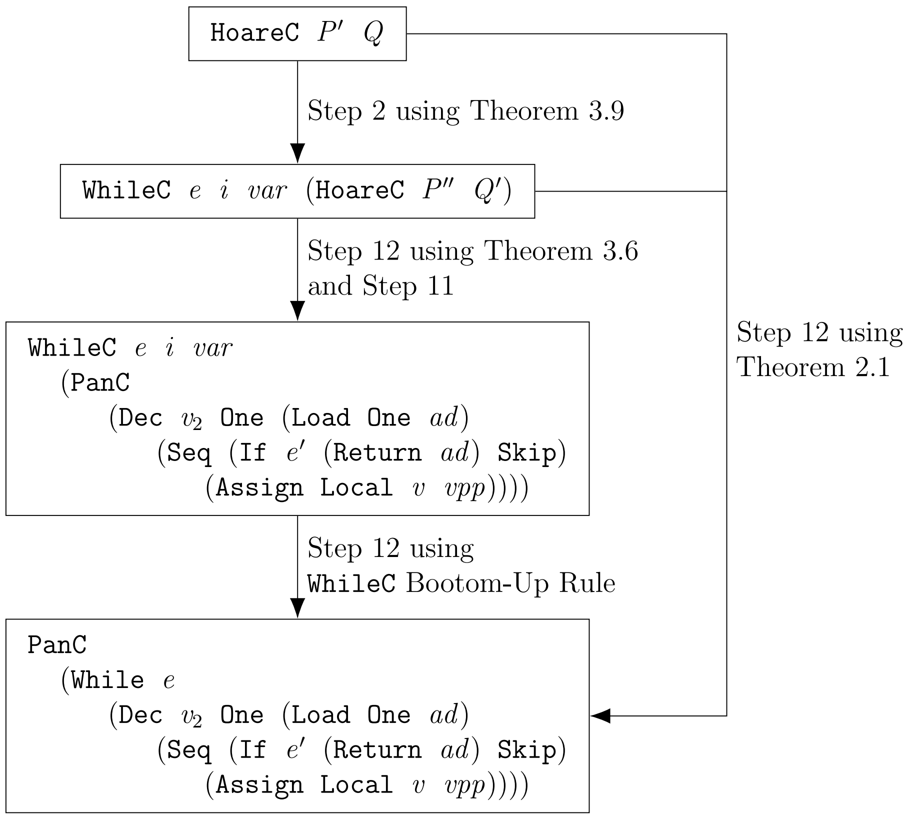
Step 1 using Theorem 3.10
Step 13 using Theorem 3.6 and Step 12
Step 13 using DecC Bootom-Up Rule
Step 13 using Theorem 2.1

Figure 4.2: Case Study: Refinement Steps 1 and 13

Step 2: While Loop

We continue by introducing the while loop. This can be done by applying the while refinement rule shown in Theorem 3.9, choosing the guard e, the invariant i, the variant var ensuring the termination, and the partial postconditions QB, QR, and QE, each implying the postcondition Q for a fixed result `Break`, `Return`, or `Exception`:

```
e = Cmp Lower (Var Local v) (Const l)
i =
(λ s.
     array_in_mem b l s ∧ s.memory = the_mem ∧
     ∃ w. var_eq_val Local v (ValWord w) s ∧ w <₊ l ∧
         appears (Word x) (b + w) (l − w) s)
var =
(λ s.
     if FLOOKUP s.locals v ≠ NONE then w2n (l − var_word v s)
     else 0)
QB = (λ s. F)
QR =
(λ (s,r).
     ∃ ret.
       r = ValWord ret ∧ b ≤₊ ret ∧ ret <₊ b + l ∧
       array_in_mem b l s ∧ s.memory = the_mem ∧
       s.memory ret = Word x)
QE = (λ (s,eid,e). F),
```


Step 2 using Theorem 3.9
Step 12 using Theorem 3.6 and Step 11
Step 12 using WhileC Bootom-Up Rule
Step 12 using Theorem 2.1

```
HoareC P' Q

WhileC e i var (HoareC P'' Q')

WhileC e i var
  (PanC
     (Dec v2 One (Load One ad)
        (Seq (If e' (Return ad) Skip)
           (Assign Local v vpp))))

PanC
  (While e
     (Dec v2 One (Load One ad)
        (Seq (If e' (Return ad) Skip)
           (Assign Local v vpp))))
```

Figure 4.3: Case Study: Refinement Steps 2 and 12

where `w2n` is the word-to-number conversion and `var_word` gets the word-value of a variable. This refinement step is shown in Figure 4.3.

Using these abbreviations, we want to prove that

```
refine (HoareC P' Q)
  (WhileC e i var
     (HoareC (while_body_pre i e)
        (while_body_post i QB QR QE))),
```

by applying the while refinement rule and proving the side conditions

```
clkfree_p P' ∧ clkfree_p i ∧ clkfree_q Q ∧
(∀t. QB t ⇒ Q (NONE,t)) ∧
(∀t v. QR (t,v) ⇒ Q (SOME (Return v),t)) ∧
(∀t eid v. QE (t,eid,v) ⇒ Q (SOME (Exception eid v),t)) ∧
(∀s. P' s ⇒ i s) ∧ (∀s. i s ⇒ evaluates_to_word e s) ∧
∀s. i s ∧ evaluates_to_false e s ⇒ Q (NONE,s),
```

i.e., the clock-freeness of precondition, invariant, and postcondition, the implication between partial postconditions and postcondition, the necessary properties of the invariant.

These goals can be solved by applying Definitions 3.2, 3.9, and 4.1 to 4.3, as well as the definitions of the predicates `evaluates_to_word`, `evaluates_to_true`, and `evaluates_to_false`.

For brevity, we refer to the precondition and postcondition of this subcontract as

```
P'' = while_body_pre i e
Q' = while_body_post i QB QR QE.
```

Step 3: Load Word from Memory

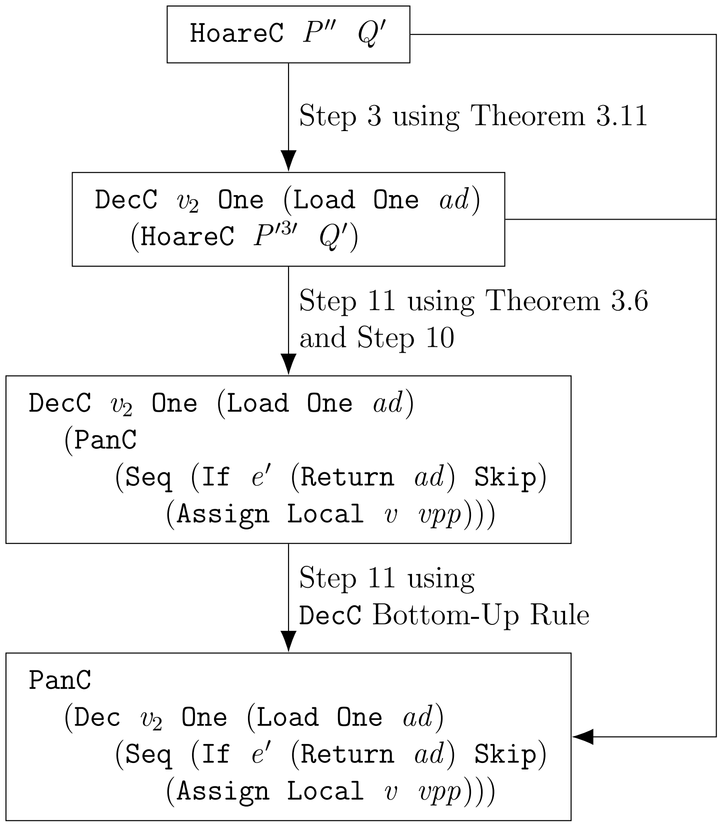

HoareC P'' Q'
Step 3 using Theorem 3.11
DecC v_2 One (Load One ad) (HoareC $P'^{3'}$ Q')
Step 11 using Theorem 3.6 and Step 10
Step 11 using Theorem 2.1
DecC v_2 One (Load One ad) (PanC (Seq (If e' (Return ad) Skip) (Assign Local v vpp)))
Step 11 using DecC Bottom-Up Rule
PanC (Dec v_2 One (Load One ad) (Seq (If e' (Return ad) Skip) (Assign Local v vpp)))

Figure 4.4: Case Study: Refinement Steps 3 and 11

We commence the loop body by loading a word from the position in the array indicated by the counter variable v. Assuming that we want to load this word to a fresh variable, this can be achieved using the declaration assignment rule for memory loads shown in Theorem 3.11. This refinement step is shown in Figure 4.4.

Abbreviating the load address as ad `= Op Add [Const` b`; Var Local` v`]`, we want to prove that

$v \neq v_2 \Rightarrow$
`refine (HoareC` P'' Q'`)`
  `(DecC` v_2 `One (Load One` ad`)`
    `(HoareC (`$\lambda s.$ P'' s $\wedge$ `var_eq_mem Local` v_2 ad `One` s`)` Q'`))`

by applying the refinement rule and proving the side conditions

`clkfree_p` P'' $\wedge$ `clkfree_q` Q' $\wedge$ `¬MEM` v_2 `(var_exp` ad`)` $\wedge$
`varfree_p` v_2 P'' $\wedge$ `varfree_q` v_2 Q' $\wedge$
$\forall s.$ P'' s $\Rightarrow$ `evaluates_shape (Load One` ad`) One` s,

i.e., the clock-freeness and variable-freeness of the precondition and postcondition, that v_2 does not appear in ad, and that the precondition implies `Load One` ad evaluating to the shape `One`.

This can be done by applying Definitions 3.2, 3.9, and 4.1 to 4.3, as well as the definitions of `varfree_p`, `varfree_q`, `evaluates_to_true`, `evaluates_shape`, and `var_exp`.

We again abbreviate the precondition of this subcontract as

$P'^{3'}$ `= (`$\lambda s.$ P'' s $\wedge$ `var_eq_mem Local` v_2 ad `One` s`).`

Step 4: Sequence for Loop Body

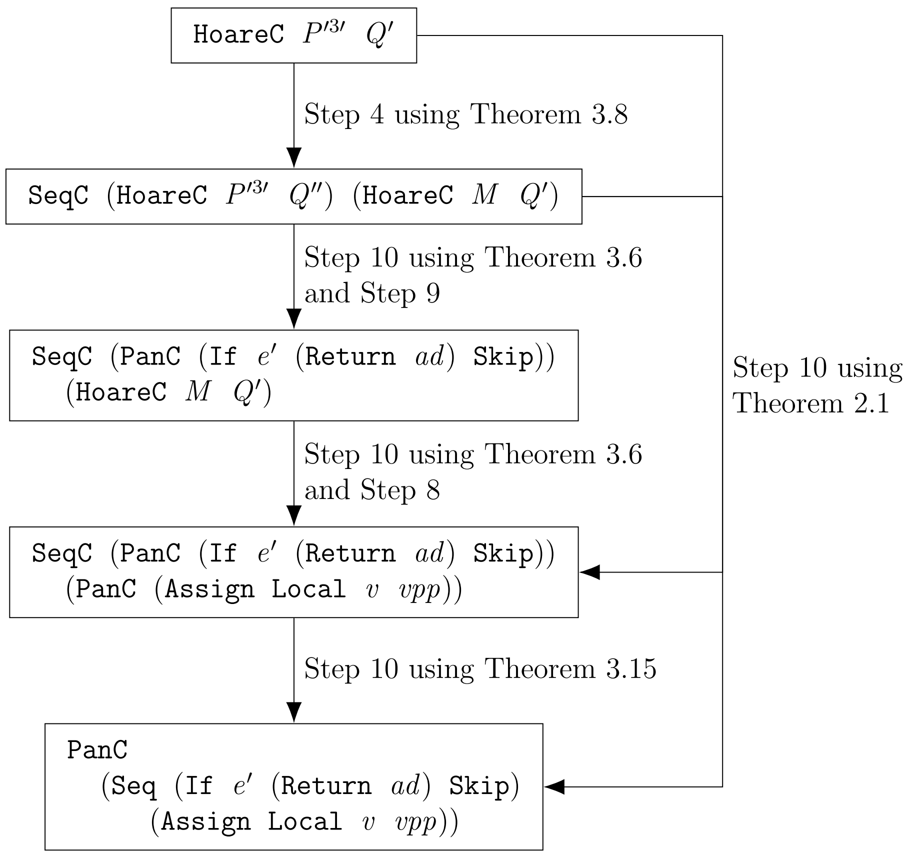

HoareC $P'^{3'}$ Q'
Step 4 using Theorem 3.8
SeqC (HoareC $P'^{3'}$ Q'') (HoareC M Q')
Step 10 using Theorem 3.6 and Step 9
SeqC (PanC (If e' (Return ad) Skip)) (HoareC M Q')
Step 10 using Theorem 3.6 and Step 8
SeqC (PanC (If e' (Return ad) Skip)) (PanC (Assign Local v vpp))
Step 10 using Theorem 3.15
PanC (Seq (If e' (Return ad) Skip) (Assign Local v vpp))
Step 10 using Theorem 2.1

Figure 4.5: Case Study: Refinement Steps 4 and 10

We want to structure the remainder of the loop body as a sequence: We first want to check if the fetched word is the sought-after word, possibly returning the address, and then we possibly want to increment the counter variable. Therefore, we need to apply a sequence refinement rule that allows the termination in both sub-programs, i.e., the rule introduced in Theorem 3.8. This refinement step is shown in Figure 4.5.

Introducing an intermediate condition

```
$M$ =
($\lambda s$.
    $P'^{3'}$ $s$ $\wedge$ $\neg$var_eq_val Local $v_2$ (ValWord $x$) $s$ $\wedge$
    $\exists w$. var_eq_val Local $v$ (ValWord $w$) $s$ $\wedge$ $w$ + 1w $<_+$ $l$),
```

we want to prove that

```
$v \neq v_2 \Rightarrow$
refine (HoareC $P'^{3'}$ $Q'$)
  (SeqC
    (HoareC $P'^{3'}$
      ($\lambda$($r$,$t$). if $r \neq$ NONE then $Q'$ ($r$,$t$) else $M$ $t$))
    (HoareC $M$ $Q'$))
```

by applying the refinement rule and proving the side conditions

```
clkfree_p $M$ $\wedge$ clkfree_p $P'^{3'}$ $\wedge$ clkfree_q $Q'$.
```

This can be done by applying Definitions 3.2, 3.9, and 4.1 to 4.3, as well as the definitions of `var_eq_val`, `var_eq_mem`, and `evaluates_to_true`.

We abbreviate the postcondition of the first subcontract as

$Q'' = (\lambda\,(r,t).\ \textbf{if}\ r \neq \texttt{NONE}\ \textbf{then}\ Q'\ (r,t)\ \textbf{else}\ M\ t)$.

Step 5: Conditional on Memory Value

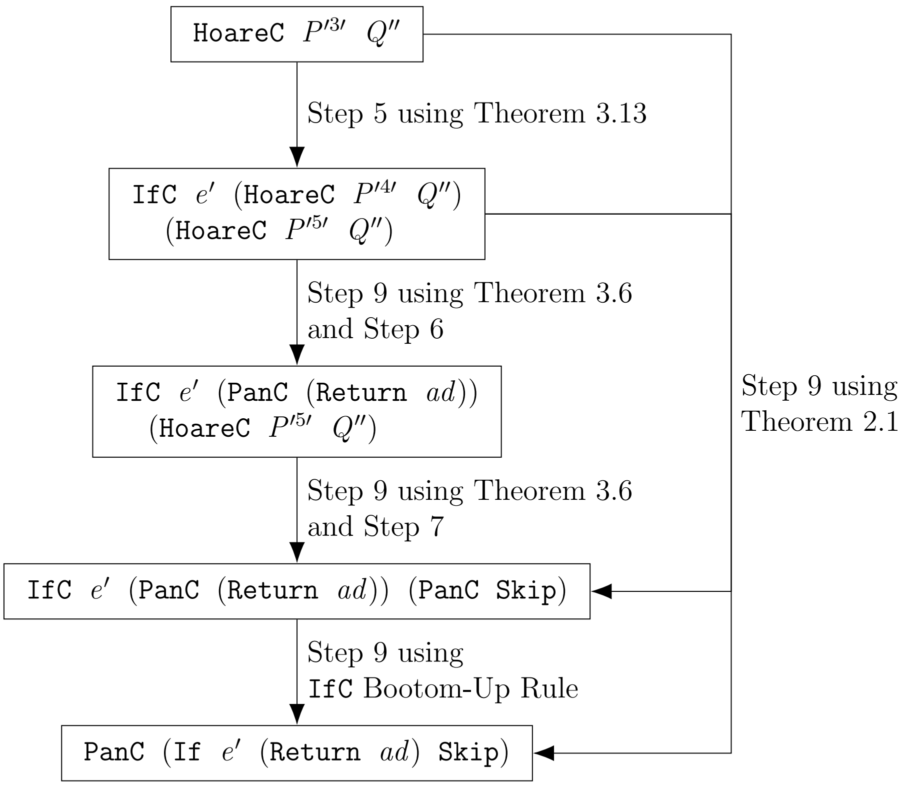

HoareC $P'^{3'}$ Q''
Step 5 using Theorem 3.13
IfC e' (HoareC $P'^{4'}$ Q'') (HoareC $P'^{5'}$ Q'')
Step 9 using Theorem 3.6 and Step 6
IfC e' (PanC (Return ad)) (HoareC $P'^{5'}$ Q'')
Step 9 using Theorem 3.6 and Step 7
IfC e' (PanC (Return ad)) (PanC Skip)
Step 9 using IfC Bootom-Up Rule
PanC (If e' (Return ad) Skip)
Step 9 using Theorem 2.1

Figure 4.6: Case Study: Refinement Steps 5 and 9

For the left-hand side of the sequence, we want to introduce a conditional on the value of variable v_2. Abbreviating e' = `Cmp Equal (Var Local` v_2`) (Const` x`)`, we use the if refinement rule shown in Theorem 3.13, proving

$v \neq v_2 \Rightarrow$
$\texttt{refine (HoareC}\ P'^{3'}\ Q'')$
$\quad\texttt{(IfC}\ e'$
$\qquad\texttt{(HoareC}\ (\lambda s.\ P'^{3'}\ s \wedge \texttt{evaluates\_to\_true}\ e'\ s)\ Q'')$
$\qquad\texttt{(HoareC}\ (\lambda s.\ P'^{3'}\ s \wedge \texttt{evaluates\_to\_false}\ e'\ s)\ Q''))$

by proving the side conditions

$\texttt{clkfree\_p}\ P'^{3'} \wedge \texttt{clkfree\_q}\ Q'' \wedge$
$\forall s.\ P'^{3'}\ s \Rightarrow \texttt{evaluates\_to\_word}\ e'\ s$,

i.e., the clock-freeness of precondition and postcondition and that the precondition ensures the evaluation of the guard to a boolean value. This refinement step is shown in Figure 4.6.

This can be done by applying Definitions 3.2, 3.9, and 4.1 to 4.3, as well as the definitions of `var_eq_val`, `var_eq_mem`, `evaluates_to_word`, and `evaluates_to_true`.

We abbreviate the preconditions of both branches as

$P'^{4'} = (\lambda s.\ P'^{3'}\ s \wedge \texttt{evaluates\_to\_true}\ e'\ s)$
$P'^{5'} = (\lambda s.\ P'^{3'}\ s \wedge \texttt{evaluates\_to\_false}\ e'\ s)$.

Step 6: Then-Branch

$$\begin{tikzcd}
\boxed{\texttt{HoareC}\ P'^{4'}\ Q''} \arrow[d, "\text{Step 6 using}\\ \text{Theorem 3.14}"] & & \boxed{\texttt{HoareC}\ P'^{5'}\ Q''} \arrow[d, "\text{Step 7 using}\\ \text{Theorem 3.7}"] \\
\boxed{\texttt{PanC (Return}\ ad\texttt{)}} & & \boxed{\texttt{PanC Skip}}
\end{tikzcd}$$

$$\begin{tikzcd}
\boxed{\texttt{HoareC}\ M\ Q'} \arrow[d, "\text{Step 8 using}\\ \text{Theorem 3.12}"] \\
\boxed{\texttt{PanC (Assign Local}\ v\ vpp\texttt{)}}
\end{tikzcd}$$

Figure 4.7: Case Study: Refinement Steps 6, 7, and 8

For the then-branch of the conditional, we want to return *ad*, which can be done by applying the return refinement rule presented in Theorem 3.14. This refinement step is shown in Figure 4.7. Therefore, we want to prove

```
$v \neq v_2 \Rightarrow$ refine (HoareC $P'^{4'}$ $Q''$) (PanC (Return $ad$))
```

by applying the return refinement rule and proving the side conditions

```
clkfree_p $P'^{4'}$ $\wedge$ clkfree_q $Q''$ $\wedge$
$\forall s.$ $P'^{4'}$ $s \Rightarrow$
    $\exists val.$
      evaluates_to $ad$ $val$ $s$ $\wedge$
      size_of_shape (shape_of $val$) $\leq 32$ $\wedge$
      $Q''$ (SOME (Return $val$),empty_locals $s$),
```

i.e., the clock-freeness of precondition and postcondition, and that the precondition implies the evaluation of *ad* to a valid return value.

This can be done by applying Definitions 3.2, 3.9, and 4.1 to 4.3, as well as the definitions of the predicates and operators `var_eq_val`, `var_eq_mem`, `evaluates_to_true`, `evaluates_to`, `size_of_shape`, `shape_of`, and `empty_locals`.

Step 7: Else-Branch

For the else-branch of the conditional, we want to apply the skip refinement rule, as this is the equivalent of a missing else branch. This refinement step is shown in Figure 4.7. Therefore, we want to prove

```
$v \neq v_2 \Rightarrow$ refine (HoareC $P'^{5'}$ $Q''$) (PanC Skip)
```

by applying the skip refinement rule and proving the side conditions

```
clkfree_p $P'^{5'}$ $\wedge$ clkfree_q $Q''$ $\wedge$ $\forall s.$ $P'^{5'}$ $s \Rightarrow$ $Q''$ (NONE,$s$),
```

i.e., the clock-freeness of precondition and postcondition and the implication between them.

This can be done by applying Definitions 3.2, 3.9, and 4.1 to 4.3, as well as the definitions of `var_eq_val`, `var_eq_mem`, `evaluates_to_true` and `evaluates_to_false`.

Step 8: Counter Increment

To increment the counter in the right-hand side of the sequence, we apply the assignment refinement rule shown in Theorem 3.12. This refinement step is shown in Figure 4.7. Abbreviating

```
vpp = Op Add [Const 1w; Var Local v],
```

we want to prove

```
v ≠ v₂ ⇒ refine (HoareC M Q′) (PanC (Assign Local v vpp))
```

by applying the assign refinement rule and proving the side conditions

```
clkfree_p M ∧ clkfree_q Q′ ∧
∀s. M s ⇒
    valid_value Local v vpp s ∧
    subst Local v vpp (λs. Q′ (NONE,s)) s,
```

i.e., the clock-freeness of precondition and postcondition, that the term *vpp* evaluates to the same shape as v, and that the state with v substituted fulfills the postcondition.

This can be done by applying Definitions 3.2, 3.9, and 4.1 to 4.3, as well as the definitions of `var_eq_val`, `var_eq_mem`, `subst`, `valid_value`, `evaluates_to_true` and `evaluates_to_false`.

This finishes the top-down refinement steps, allowing us to continue with the bottom-up steps.

Steps 9 to 13: Bottom-Up Steps

The bottom-up steps 9 to 13 are all structured similarly, each incorporating the application of monotonicity (Theorem 3.6), transitivity (Theorem 2.1), and bottom-up refinement rules. For brevity, we only describe Step 12, i.e., the bottom-up step for the while loop. This refinement step is particularly interesting, as it contains the termination proof of the loop. The Steps 9 to 11 and 13 can be traced by inspecting Figures 4.2 to 4.7, in which the bottom-up refinement steps are each visualized with their corresponding top-down refinement steps.

Beginning with the proof of Step 12, we know from Step 11 that

```
v ≠ v₂ ⇒
refine (HoareC P″ Q′)
  (PanC
     (Dec v₂ One (Load One ad)
        (Seq (If e′ (Return ad) Skip) (Assign Local v vpp)))).
```

By applying Theorem 3.6, it holds that

```
v ≠ v₂ ⇒
refine (WhileC e i var (HoareC P″ Q′))
  (WhileC e i var
     (PanC
        (Dec v₂ One (Load One ad)
           (Seq (If e′ (Return ad) Skip)
              (Assign Local v vpp))))).
```

To apply the bottom-up refinement rule for while loops shown in Theorem 3.16, we have to show that

```
v ≠ v2 ⇒
is_variant i var
  (Dec v2 One (Load One ad)
     (Seq (If e′ (Return ad) Skip) (Assign Local v vpp)))
```

holds. This can be done by applying Definition 3.6, followed by the application of the definition of `evaluate` shown in Appendix C and multiple goal-oriented case splits.

Applying the bottom-up refinement rule for while loops yields

```
v ≠ v2 ⇒
refine
  (WhileC e i var
     (PanC
        (Dec v2 One (Load One ad)
           (Seq (If e′ (Return ad) Skip)
              (Assign Local v vpp)))))
  (PanC
     (While e
        (Dec v2 One (Load One ad)
           (Seq (If e′ (Return ad) Skip)
              (Assign Local v vpp))))),
```

and by transitivity (Theorem 2.1), it holds that

```
v ≠ v2 ⇒
refine (WhileC e i var (HoareC P″ Q′))
  (PanC
     (While e
        (Dec v2 One (Load One ad)
           (Seq (If e′ (Return ad) Skip)
              (Assign Local v vpp))))).
```

Using Step 2 and transitivity (Theorem 2.1) yields

```
v ≠ v2 ⇒
refine (HoareC P′ Q)
  (PanC
     (While e
        (Dec v2 One (Load One ad)
           (Seq (If e′ (Return ad) Skip)
              (Assign Local v vpp))))).
```

4.3 Automating Refinement Proofs

As it is visible in Section 4.2, the single refinement steps share a common proof structure. We use this property to provide users of our calculus with a HOL4 tactic which is able to solve most of the proof goals arising in a refinement proof. In this section, we describe the general proof scheme encapsuled in this tactic, focussing on top-down refinement steps in Section 4.3.1 and on bottom-up refinement steps in Section 4.3.2. In both sections, we discuss the limitations of our proof tactic and provide suggestions for improvements. The HOL4 implementations of our tactics can be found in Section D.5.

4.3.1 Top-Down Refinement Steps

For top-down refinement steps, i.e., refinement steps introducing new constracts from `HoareC` contracts, we provide a parameterized proof tactic which takes a refinement rule and a set of theorems as arguments. The proof tactic consists of two major steps: (1) matching the provided refinement rule with the proof goal and (2) proving the side conditions by the application of definitions and theorems from our calculus, as well as the ones provided for the tactic application. To explain the process in detail, we use Step 6 of the preceding case study as a running example for this subsection.

For rule matching (1), we ensure a successful match of a refinement rule by requiring the user to provide suitable abbreviations, if a precondition P or a postcondition Q appears as a subterm on both sides of a refinement. This can be done using tactics from HOL4's *bossLib*, e.g., `qabbrev_tac`, which abbreviates a term occurring in the proof goal if it matches the provided pattern. As in our example

$\texttt{refine (HoareC } P'^{4\prime}\ Q''\texttt{) (PanC (Return } ad\texttt{))}$

does not contain a condition on both sides of the refinement, no abbreviation is needed.

Once the goal matches the refinement rule, one may use `irule` to match the right-hand side of the refinement rule with the goal and to replace it with the side conditions

$$\begin{array}{l}\texttt{clkfree\_p}\ P'^{4\prime} \wedge \texttt{clkfree\_q}\ Q'' \wedge \\ \forall s.\ P'^{4\prime}\ s \Rightarrow \\ \quad \exists val. \\ \qquad \texttt{evaluates\_to}\ ad\ val\ s \wedge \\ \qquad \texttt{size\_of\_shape (shape\_of}\ val\texttt{)} \leq 32 \wedge \\ \qquad Q''\ \texttt{(SOME (Return}\ val\texttt{),empty\_locals}\ s\texttt{)},\end{array}$$

After this step, the remaining goal is unabbreviated to allow proving the side conditions.

For subgoal proving (2), we provide the set of relevant definitions and theorems of our calculus as a HOL4 *simpset*, which is combined with the user-supplied theorems. The resulting set is applied to the side conditions using repeated use of `rw_tac` until the goal remains unchanged. In our example, this yields the goal depicted below and two clock-freeness goals.

```
 0.  v <> v2
     [...]
10.  (case
        OPTION_BIND (OPTION_BIND (FLOOKUP s.locals v)
          (\ h. SOME [h])) (\ t. SOME (ValWord b::t))
      of
        [...]) = SOME (ValWord addr')
11.  s.memaddrs addr'
12.  FLOOKUP s.locals v2 = SOME (Val (s.memory addr'))
13.  (case FLOOKUP s.locals v2 of
        [...]) = SOME (ValWord w'')
14.  w'' <> 0w
------------------------------------
     b <=+ addr' /\ addr' <+ b + 1 /\ s.memory addr' = Word x
```

Next, we try to instantiate possibly occurring existential subgoals as the goal above by using `HINT_EXISTS_TAC`, which matches existential quantifiers with terms from the assumptions. In our example, there is quantifier to instantiate.

The goals remaining after this step often contain terms allowing case splits, e.g., the assumptions `10` and `13` in the example above, most of them leading to contradictions. Therefore, we continue with repeated applications of `every_case_tac`, splitting these terms and producing multiple subgoals, each followed by an application our refinement *simpset*, until the goal is unchanged. In our example, this yields

```
    0.  v <> v2
    1.  b <+ b + 1
    2.  !addr. b <=+ addr / addr <+ b + 1 ==> s.memaddrs addr
    3.  FLOOKUP s.locals v = SOME (ValWord w)
    4.  w <+ 1
    5.  b + w <=+ addr
    6.  addr <+ b + 1
        [...]
  ------------------------------------
        b <=+ b + w /\ b + w <+ b + 1
```

After this step, the subgoals occurring for our case study can be distinguished in three groups: (1) goals of first-order logic, where an assumption has not been matched with the goal, (2) clock-freeness goals, and (3) goals of word arithmetic.

The goals of type (1) can be solved by matching universally quantified assumptions with the goal. In our example, there is no goal of this type.

The goals of type (2) are clock-freeness goals of conditions, which could not be shown to be a composition of other clockfree conditions using theorems from Section D.1. In this case, it is sufficient to apply Definition 3.2 and theorem 2.5 with the simplifier `gvs`. In our example, this is necessary for both $P'^{4'}$ and Q''.

The goals of type (3) which mostly arise from word arithmetic inequalities, e.g., the goal provided in the example above, generally could be solved by strategic application of theorems from HOL4's *wordsTheory*. In the course of this thesis, we were not able to provide an automated strategy based on *wordsTheory* which is sufficient for all goals arising in our case study. Instead, we decided to use HOL4's *blastLib* to solve the goals for a given word size, i.e., by converting the proof goal to propositions about a word's bits and solving them using a SAT solver.

For our case study, *blastLib* was able to solve these goals, but took up to 20 seconds for subgoals with a large number of assumptions. To improve the usability of our automated proof tactic, we additionally implemented an amended version of our tactic, in which we replaced *blastLib* with a call to the SMT solver *Z3* [MB08] using `z3o_tac` from HOL4's *HolSmtLib*. This usually allows us to decrease the completion time to less than a second by giving up end-to-end correctness, as the conversions within `z3o_tac` are not verified. We ruled out to use the verified `z3_tac`, as the included verification of Z3 proof scripts is even slower than using *blastLib*.

Developers using our automated proof tactic might choose between *HolSmtLib* and *blastLib*, considerating the required execution time and reliability,

4.3.2 Bottom-Up Refinement Steps

For bottom-up refinement steps, we can distinguish three cases, as our case study in Section 4.2 shows: (1) steps by transitivity, i.e., using Theorem 2.1, (2) steps by monotonicity, i.e., using Theorem 3.6, and (3) steps by bottom-up refinement rules.

Both (1) and (2) are trivially proven by initializing both Theorem 2.1 and Theorem 3.6 with the results from other refinement steps. For (3) in all cases except the bottom-up rule for while loops, it is sufficient to apply the refinement rule, e.g., using a built-in simplifier like `gvs`. For while loops, the application of the refinement rule (Theorem 3.16) is not sufficient, as it requires proving the variant.

For examples smaller than our case study, we were able to show that applying our tactic from Section 4.3.1, i.e., repeated application of definitions and theorems, each followed by case splits, is sufficient. For our case study, the body of the loop contains a large number of cases, causing the repeated case splits to combinatorically explode. To still complete the proofs for our case study within decent time, we manually choose goal-directed case splits which allow a quick termination.

Limited by the completion time of this thesis, we are not able to provide an automated approach solving this issue. For future work, we see two possible approaches for a solution: (1) selecting case splits more carefully than using `every_case_tac`, e.g., with an outside-in approach, or (2) mitigating the issue completely by integrating the variant into the sub-contract of a `WhileC`, simplifying the bottom-up rule, but possibly requiring larger adaptions in the remainder of the calculus.

5. Related Work

In this chapter, we discuss existing research with relevance to the topic of this thesis. As the work on CbC and refinement-based programming as well as Pancake is discussed in detail in Chapter 2, we focus on two research areas close to our contribution: (1) the formalization of refinement concepts like CbC using theorem provers, and (2) the verification of systems software, including verified microkernels and verification tools for systems programming languages.

Refinement Concepts in Theorem Provers

Back and Wright formalize Back's refinement calculus in *HOL88*, a precursor of *HOL4* [BW90]. Using the GCL statements' WP as formal semantics of the programming language, they define a refinement relation between two programs as the implication between their WPs, allowing them to prove program-to-program refinement rules. In contrast to our contribution, they do not discuss refinements between specifications and programs.

Von Wright et al. extend the preceding contribution by formalizing additional refinement concepts, e.g., the refinement between data structures [Von+93]. Additionally, they rephrase the calculus as a calculus of predicate transformers, enabling the refinement of a specification to a program. In their conclusion, they state that a better user interface than HOL is necessary to allow the use of their implementation on larger programs.

Based on the contributions above, Butler and Långbacka provide a graphical user interface to provide support for refinement proofs [BL96]. This interface is implemented as an extension to the HOL window library *TkWinHOL*, adding refinement-specific features. It especially allows the user to perform sub-derivations, i.e., using the monotonicity of program construction, by selecting sub-terms with the mouse.

More recently, Alpuim and Swierstra implement the refinement calculus as described by Morgan in the theorem prover *Coq* [AS18]. As the contributions above, they implement the refinement relation as a relation on predicate transformers but add a special value SPEC as a program, indicating a specification. This resembles Morgan's terminology discussed in Section 2.1.1 and is similar to our approach with our *contracts* being equivalent to Alpuim's and Swierstra's *programs* who provide predicates on programs to destinguish between specifications and programs in our sense.

Verification of Systems Software

For the verification of systems software, we discuss two kinds of existing work: (1) contributions towards verified kernels and operating systems, and (2) verification techniques and tools for systems programming languages. All discussed contributions focus on post-hoc verification techniques, in contrast to our implementation of an a priori approach.

Klein provides an overview about the state of the art in verified kernels and operating systems in 2009 [Kle09]. As the first steps towards a verified operation system, Klein lists the developments of *UCLA Secure Data Unix* [WKP80], the *Provable Secure Operating System (PSOS)* [FN79], and the *Kernel for Isolated Tasks (KIT)* [Bev89]. For both *UCLA Secure Data Unix* and *PSOS*, the correctness proofs were not finished, making *KIT* the first verified kernel, with its features limited to task isolation, asynchronous I/O, exception handling, and single-word message passing.

For the more recent approaches on scaling kernel verification, Klein names the *seL4* and *L4.verified* projects which focus on a new implementation of the *L4* microkernel and its verification, resulting in the later *seL4* [Kle+14]. After the completion of the kernel correctness proofs in 2009, there have been multiple contributions proving properties of *seL4*, including the correctness of its machine code and the security properties of integrity and confidentiality. The kernel *seL4* and especially its driver framework, *sDDF*[1], are closely related to our contribution, as Pancake is targeted to be used in *sDDF* for driver development and used as the target programming language of our CbC calculus.

For verification tools for systems programming languages, Klein lists the *Verisoft* project which implements an end-to-end post-hoc verification approach from application to gate level [Alk+08], strongly focussing on verification of the different compiler steps. It targets systems programs written in the C-subsets *C0* and *C0A* and uses verified compilation to machine code for the VAMP microprocessor.

A larger subset of C is supported by the post-hoc verification tool *VCC* which was implemented in the successor project of *Verisoft* [Coh+09]. It performs correctness proofs for concurrent C programs by transpiling them and their specifications into the intermediate language *Boogie*, followed by a call to the *Boogie* program verifier. This verifier calls the SMT solver *Z3* which we use within this thesis for automatic refinement proofs.

Besides tools for C, the most notable verification tool is *Verus*, a post-hoc verification tool for Rust [Lat+23]. As *VCC*, *Verus* is based on a SMT solver but strongly relies on the properties of Rust's typesystem, i.e., linear types and borrow checking.

[1]https://trustworthy.systems/projects/drivers/

6. Conclusion

Operating systems and drivers are prevalent in safety-critical environments and can thus profit from formal verification. Current verification approaches for low-level software are limited to post-hoc techniques, leaving potential for improvement through an a priori, refinement-based approach.

In this thesis, we introduced a CbC calculus for a subset of the systems programming language *Pancake*. Our calculus allows the a priori correct implementation of Pancake programs, using correctness-preserving refinement steps from Hoare-style specifications into Pancake programs. It supports a Turing-complete subset of the programming language, consisting of 15 statement types which we introduce in Section 2.2. For these 15 statement types, we provided 22 top-down refinement rules, listed in Section D.4, each dealing with the introduction of a statement type, possibly distinguishing multiple cases in the WP of the statement type. Additionally, we provided four bottom-up refinement rules for the compositional statement types which can be used to transform compositional contracts requiring programs into program.

To demonstrate the feasibility of our approach, we provided a case study in Chapter 4 in which we implemented linear search in Pancake using our calculus. To allow developers without detailled knowledge of *HOL4* to use our calculus, we implemented an automated proof tactic which is able to solve most proof goals occurring in our case study, with the relevant limitation of while loop variant proofs.

Future Work

For future work on CbC for systems programming, we plan to investigate the following leads: Initially, we plan to extend our CbC calculus to the remaining statement types of Pancake, i.e., adding shared memory and function calls. The former requires new user-facing syntax for preconditions and postconditions which allows the simple specification of interactions with Pancake's *foreign function interface*. The latter requires reasoning about the termination of recursion which might be approached as while loops, i.e., using variants.

Secondly, we intend to restructure the inclusion of while loops in our calculus, as the current solution of postponing the termination proof introduces problems for the automated proof of refinement steps. Therefore, our goal is to integrate the loop variant into the Hoare contract for the loop body, but the implications of this change on the remainder of the calculus should be studied carefully.

Thirdly, we plan to continue the work towards an automated proof tactic which is able to deal with all real-world refinement steps, especially with proving loop variants which might also be mitigated by solving the issue above. To demonstrate the feasibility of the automated proof tactic, it is necessary to provide multiple larger case studies which use all statement types and a multitude of specification styles. Additionally, it would be particularly interesting to enhance automated proofs for word-arithmetic goals, possibly removing Z3 from our dependencies.

At last, we aim to integrate our approach into more user-friendly CbC tool support, e.g., into the CorC [Bor+23] successor, which is currently under development. Using an enhanced automated proof tactic, it might be possible to add a new backend to the existing software which creates HOL4 proof scripts from user-provided top-down refinement applications and automatically selected bottom-up rules applications.

Bibliography

[Alk+08] Eyad Alkassar et al. “The Verisoft Approach to Systems Verification”. In: *Verified Software: Theories, Tools, Experiments*. Ed. by Natarajan Shankar and Jim Woodcock. Berlin, Heidelberg: Springer, 2008, pp. 209–224. ISBN: 978-3-540-87873-5. DOI: 10.1007/978-3-540-87873-5_18.

[AS18] João Alpuim and Wouter Swierstra. “Embedding the refinement calculus in Coq”. In: *Science of Computer Programming*. Special issue of selected papers from FLOPS 2016 164 (Oct. 15, 2018), pp. 37–48. ISSN: 0167-6423. DOI: 10.1016/j.scico.2017.04.003.

[Bac81] R. J. R. Back. “On correct refinement of programs”. In: *Journal of Computer and System Sciences* 23.1 (Aug. 1, 1981), pp. 49–68. ISSN: 0022-0000. DOI: 10.1016/0022-0000(81)90005-2.

[Bac88] R. J. R. Back. “A calculus of refinements for program derivations”. In: *Acta Informatica* 25.6 (Aug. 1, 1988), pp. 593–624. ISSN: 1432-0525. DOI: 10.1007/BF00291051.

[Bev89] W.R. Bevier. “Kit: a study in operating system verification”. In: *IEEE Transactions on Software Engineering* 15.11 (Nov. 1989), pp. 1382–1396. ISSN: 1939-3520. DOI: 10.1109/32.41331.

[BL96] Michael Butler and Thomas Långbacka. “Program derivation using the refinement calculator”. In: *Theorem Proving in Higher Order Logics*. Ed. by Gerhard Goos et al. Berlin, Heidelberg: Springer, 1996, pp. 93–108. ISBN: 978-3-540-70641-0. DOI: 10.1007/BFb0105399.

[Bor+23] Tabea Bordis et al. “Correctness-by-Construction: An Overview of the CorC Ecosystem”. In: *Ada Lett.* 42.2 (Apr. 5, 2023), pp. 75–78. ISSN: 1094-3641. DOI: 10.1145/3591335.3591343.

[BW90] R. J. R. Back and J. von Wright. “Refinement concepts formalised in higher order logic”. In: *Form. Asp. Comput.* 2.1 (Mar. 1, 1990), pp. 247–272. ISSN: 0934-5043. DOI: 10.1007/BF01888227.

[Car+98] D. Carrington et al. “A Program Refinement Tool”. In: *Formal Aspects of Computing* 10.2 (Nov. 1998). Publisher: Association for Computing Machinery (ACM), pp. 97–124. ISSN: 0934-5043, 1433-299X. DOI: 10.1007/s001650050006.

[Coh+09] Ernie Cohen et al. “VCC: A Practical System for Verifying Concurrent C”. In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Berlin, Heidelberg: Springer, 2009, pp. 23–42. ISBN: 978-3-642-03359-9. DOI: 10.1007/978-3-642-03359-9_2.

[Dij68] Edsger W. Dijkstra. “A constructive approach to the problem of program correctness”. In: *BIT Numerical Mathematics* 8.3 (Sept. 1, 1968), pp. 174–186. ISSN: 1572-9125. DOI: 10.1007/BF01933419.

[Dij75] Edsger W. Dijkstra. “Guarded commands, nondeterminacy and formal derivation of programs”. In: *Commun. ACM* 18.8 (Aug. 1, 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975.

[FN79] Richard J. Feiertag and Peter G. Neumann. “The foundations of a provably secure operating system (PSOS)”. In: *1979 International Workshop on Managing Requirements Knowledge (MARK).* 1979 International Workshop on Managing Requirements Knowledge (MARK). ISSN: 2164-0149. June 1979, pp. 329–334. DOI: 10.1109/MARK.1979.8817256.

[II15] International Organization for Standardization and International Electrotechnical Commission. *ISO/IEC 2382:2015 - Information technology.* Version 2015. Geneva, Switzerland, May 2015.

[Int10] International Electrotechnical Commission. *IEC 61508:2010 - Functional safety of electrical/electronic/programmable electronic safety-related systems.* Version 2010. Geneva, Switzerland, Apr. 2010.

[Kle+14] Gerwin Klein et al. “Comprehensive formal verification of an OS microkernel”. In: *ACM Transactions on Computer Systems* 32.1 (Feb. 2014), pp. 1–70. ISSN: 0734-2071, 1557-7333. DOI: 10.1145/2560537.

[Kle09] Gerwin Klein. “Operating system verification—An overview”. In: *Sadhana* 34.1 (Feb. 1, 2009), pp. 27–69. ISSN: 0973-7677. DOI: 10.1007/s12046-009-0002-4.

[Kum+14] Ramana Kumar et al. “CakeML: a verified implementation of ML”. In: *SIGPLAN Not.* 49.1 (Jan. 8, 2014), pp. 179–191. ISSN: 0362-1340. DOI: 10.1145/2578855.2535841.

[KW12] Derrick G. Kourie and Bruce W. Watson. *The Correctness-by-Construction Approach to Programming.* Berlin, Heidelberg: Springer, 2012. ISBN: 978-3-642-27918-8 978-3-642-27919-5. DOI: 10.1007/978-3-642-27919-5.

[Lat+23] Andrea Lattuada et al. “Verus: Verifying Rust Programs using Linear Ghost Types”. In: *Software Artifact (virtual machine, pre-built distributions) for ”Verus: Verifying Rust Programs using Linear Ghost Types”* 7 (OOPSLA1 Apr. 6, 2023), 85:286–85:315. DOI: 10.1145/3586037.

[MB08] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[Mor94] Carroll Morgan. *Programming from specifications (2nd ed.)* GBR: Prentice Hall International (UK) Ltd., Aug. 1994. 332 pp. ISBN: 978-0-13-123274-7.

[Owe+16] Scott Owens et al. “Functional Big-Step Semantics”. In: *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632.* Berlin, Heidelberg: Springer-Verlag, Apr. 2, 2016, pp. 589–615. ISBN: 978-3-662-49497-4. DOI: 10.1007/978-3-662-49498-1_23.

[Poh+23] Johannes Åman Pohjola et al. “Pancake: Verified Systems Programming Made Sweeter”. In: *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*. PLOS ’23. New York, NY, USA: Association for Computing Machinery, Oct. 23, 2023, pp. 1–9. ISBN: 979-8-4007-0404-8. DOI: 10.1145/3623759.3624544.

[Run+19] Tobias Runge et al. “Tool Support for Correctness-by-Construction”. In: *Fundamental Approaches to Software Engineering*. Ed. by Reiner Hähnle and Wil van der Aalst. Cham: Springer International Publishing, 2019, pp. 25–42. ISBN: 978-3-030-16722-6. DOI: 10.1007/978-3-030-16722-6_2.

[SN08] Konrad Slind and Michael Norrish. “A Brief Overview of HOL4”. In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer, 2008, pp. 28–32. ISBN: 978-3-540-71067-7. DOI: 10.1007/978-3-540-71067-7_6.

[Von+93] J. Von Wright et al. “Mechanizing some advanced refinement concepts”. In: *Formal Methods in System Design* 3.1 (Aug. 1, 1993), pp. 49–81. ISSN: 1572-8102. DOI: 10.1007/BF01383984.

[WKP80] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. “Specification and verification of the UCLA Unix security kernel”. In: *Commun. ACM* 23.2 (Feb. 1, 1980), pp. 118–131. ISSN: 0001-0782. DOI: 10.1145/358818.358825.

A. HOL4 Proof Example

To demonstrate proofs in HOL4, we prove that the factorial of all natural numbers $n \geq 2$ is even, i.e., in HOL4 notation:

```
!n.  n >= 2 ==> EVEN (fact n)
```

The definition of `fact` is given in Figure 2.7.

We begin the proof by applying the induction tactic `Induct`, yielding two subgoals, a basis subgoal for `n = 0` and the induction step. Multiple subgoals are treated as a stack in HOL with the first subgoal being the one displayed at the bottom.

```
0.  n >= 2 ==> EVEN (fact n)
------------------------------------
    SUC n >= 2 ==> EVEN (fact (SUC n))

    0 >= 2 ==> EVEN (fact 0)
```

The second of these goals, the one at the top, is a goal with an assumption, i.e., a temporary theorem that holds in the context of this subgoal which can be used to prove it.

For both subgoals, we continue with rewriting with the definition of the factorial, using one of the builtin simplification tactics, `rw`. The basic subgoal is solved as the antecedent of the implication is false. For the induction step, we receive a new subgoal.

```
0.  n >= 2 ==> EVEN (fact n)
1.  SUC n >= 2
------------------------------------
    EVEN (SUC n * fact n)
```

To allow the use of the assumption `0`, we make a case split on `n >= 2`, yielding two subgoals.

```
0.  n >= 2 ==> EVEN (fact n)
1.  SUC n >= 2
2.  ~(n >= 2)
------------------------------------
    EVEN (SUC n * fact n)

0.  n >= 2 ==> EVEN (fact n)
1.  SUC n >= 2
2.  n >= 2
------------------------------------
    EVEN (SUC n * fact n)
```

We now leave the first subgoal unchanged by applying `ALL_TAC` while suggesting that `n = 1` for the second subgoal by applying `‘n = 1‘ by gvs[]`, where `gvs` is a builtin simplification tactic. This yields two new subgoals.

```
0.  n >= 2 ==> EVEN (fact n)
1.  SUC n >= 2
2.  ~(n >= 2)
3.  n = 1
------------------------------------
    EVEN (SUC n * fact n)

0.  n >= 2 ==> EVEN (fact n)
1.  SUC n >= 2
2.  n >= 2
------------------------------------
    EVEN (SUC n * fact n)
```

For both subgoals, we apply the theorem `EVEN_MULT`:

$\vdash \texttt{EVEN}\ (m \times n) \iff \texttt{EVEN}\ m \lor \texttt{EVEN}\ n$

More precisely, we continue by matching the right-to-left implication's consequent with our goal and by replacing the goal by this implication's antecedent. Therefore, we use `iffRL` to get the implication and use `irule` for matching and replacing, yielding two new subgoals.

```
0.  n >= 2 ==> EVEN (fact n)
1.  SUC n >= 2
2.  n >= 2
------------------------------------
    EVEN (SUC n) \/ EVEN (fact n)

0.  n >= 2 ==> EVEN (fact n)
1.  SUC n >= 2
2.  ~(n >= 2)
3.  n = 1
------------------------------------
    EVEN (SUC n) \/ EVEN (fact n)
```

Finally, both goals are solved by applying the simplification tactic `gvs`. For the first goal, it concludes that `SUC 1` is `2`, which is even, which solves the goal. For the second goal, it matches assumption `2` with the antecedent in assumption `0`. The assumption's consequent solves the goal.

Combining these steps with the tacticals `>>` and `>|`, also called `THEN` and `THENL`, yields the proof shown in Figure 2.8. The tactical `>>` sequences tactics, applying them to all current subgoals. The tactical `>|` sequences tactics as well but takes a list of tactics as the second argument, containing one tactic per subgoal.

B. CbC for GCL in HOL4

In this chapter, we prove the correctness of the existing Correctness-by-Construction (CbC) calculus for Dijkstra's Guarded Command Language (GCL) using the interactive theorem prover *HOL4*. We adapt the calculus presented by Kourie and Watson [KW12], making the simplification to always select the first the first matchig guard of a guarded command list, allowing us to define a deterministic evaluation function. In contrast to Chapter 2, we assume Hoare specifications as specifications of total correctness, i.e., enforcing the termination of resulting programs. This is achieved by defining the semantics as a *functional big-step semantics* [Owe+16] as discussed for *Pancake* in Section 2.2.2.

As all proofs within this chapter are developed in *HOL4*, we omit the detailled proof scripts, only providing the proved theorems.

To structure this chapter as done for *Pancake* in Appendix D, we begin by defining the syntax and semantics of GCL in Section B.1 and extract the WPs of GCL statements from these definitions in Section B.2. In Section B.3, we continue by defining a language of contracts and a refinement relation, as well as proving properties of refinement, and end in Section B.4 with the well-known GCL refinement rules.

B.1 Syntax and Semantics of GCL

We define the syntax of GCL as the following algebraic data type:

Definition B.1 (Syntax of GCL).

```
Op = Add | Sub | Mul | Div | Eq | Less | LessEq
Expr = Var string | BinOp Expr Op Expr | Const int
Stmt =
    Skip
  | Assign string Expr
  | Seq Stmt Stmt
  | If ((Expr × Stmt) list)
  | While ((Expr × Stmt) list)
```

As Pohjola et al. do for *Pancake* [Poh+23], we define the semantics of GCL as *functional big-step semantics* [Owe+16], assuming integer-valued variables and deterministic evaluation. Initially, we define program states as the following record data type:

Definition B.2 (GCL Semantics: Program States).

$$\mathit{state} = \texttt{<|}\ \mathit{vars} : \mathit{string} \mapsto \mathit{int};\ \mathit{clock} : \mathit{num}\ \texttt{|>}$$

We continue by defining the evaluation of expressions, evaluating uninitialized variables to 0, and using 0 and 1 as integer-values for *false* and *true*:

Definition B.3 (GCL Semantics: Evaluation of Expressions).

$$\begin{aligned}
&\mathit{eval}\ (\mathit{Var}\ v)\ s \overset{\text{def}}{=} \textbf{case}\ \mathit{FLOOKUP}\ s.\mathit{vars}\ v\ \textbf{of}\ \mathit{NONE} \Rightarrow 0\ /\ \mathit{SOME}\ c \Rightarrow c\\
&\mathit{eval}\ (\mathit{BinOp}\ e_1\ \mathit{Add}\ e_2)\ s \overset{\text{def}}{=} \mathit{eval}\ e_1\ s + \mathit{eval}\ e_2\ s\\
&\mathit{eval}\ (\mathit{BinOp}\ e_1\ \mathit{Sub}\ e_2)\ s \overset{\text{def}}{=} \mathit{eval}\ e_1\ s - \mathit{eval}\ e_2\ s\\
&\mathit{eval}\ (\mathit{BinOp}\ e_1\ \mathit{Mul}\ e_2)\ s \overset{\text{def}}{=} \mathit{eval}\ e_1\ s \times \mathit{eval}\ e_2\ s\\
&\mathit{eval}\ (\mathit{BinOp}\ e_1\ \mathit{Div}\ e_2)\ s \overset{\text{def}}{=} \mathit{eval}\ e_1\ s\ /\ \mathit{eval}\ e_2\ s\\
&\mathit{eval}\ (\mathit{BinOp}\ e_1\ \mathit{Eq}\ e_2)\ s \overset{\text{def}}{=}\\
&\quad \textbf{if}\ \mathit{eval}\ e_1\ s = \mathit{eval}\ e_2\ s\ \textbf{then}\ 1\ \textbf{else}\ 0\\
&\mathit{eval}\ (\mathit{BinOp}\ e_1\ \mathit{Less}\ e_2)\ s \overset{\text{def}}{=}\\
&\quad \textbf{if}\ \mathit{eval}\ e_1\ s < \mathit{eval}\ e_2\ s\ \textbf{then}\ 1\ \textbf{else}\ 0\\
&\mathit{eval}\ (\mathit{BinOp}\ e_1\ \mathit{LessEq}\ e_2)\ s \overset{\text{def}}{=}\\
&\quad \textbf{if}\ \mathit{eval}\ e_1\ s \le \mathit{eval}\ e_2\ s\ \textbf{then}\ 1\ \textbf{else}\ 0\\
&\mathit{eval}\ (\mathit{Const}\ c)\ s \overset{\text{def}}{=} c
\end{aligned}$$

We prove that the evaluation of expressions is clock-independent:

Theorem B.1 (Evaluation of Expressions is Clock-Independent).

$$\vdash \mathit{eval}\ e\ (s\ \mathit{with}\ \mathit{clock} := k) = \mathit{eval}\ e\ s$$

For the deterministic selection of guarded commands, we define the following abbreviation in which we find the first matching guard, assuming positive integers to represent *true*:

Definition B.4 (GCL Semantics: Find First Matching Guard).

$$\begin{aligned}
&\mathit{find\_first}\ []\ s \overset{\text{def}}{=} \mathit{NONE}\\
&\mathit{find\_first}\ (h::es)\ s \overset{\text{def}}{=}\\
&\quad \textbf{if}\ \mathit{eval}\ (\mathit{FST}\ h)\ s > 0\ \textbf{then}\ \mathit{SOME}\ h\ \textbf{else}\ \mathit{find\_first}\ es\ s
\end{aligned}$$

We prove that finding the first matching guard is clock-independent:

Theorem B.2 (Finding the First Maching Guard is Clock-Independent).

$$\vdash \mathit{find\_first}\ es\ (s\ \mathit{with}\ \mathit{clock} := k) = \mathit{find\_first}\ es\ s$$

For decrementing the clock, we define the following abbreviation:

Definition B.5 (GCL Semantics: Decrementing the Clock).

$$\mathit{dec\_clock}\ s \overset{\text{def}}{=} s\ \mathit{with}\ \mathit{clock} := s.\mathit{clock} - 1$$

As results for the evaluation of statements, we define the following data type, in which `OK` represents normal termination, `Abort` abnormal termination due to no matching guard in a selection statement, and `TimeOut` the termination due to a timeout ot the clock:

Definition B.6 (GCL Semantics: Evaluation Results).

$$\mathit{result} = \mathit{OK}\ /\ \mathit{Abort}\ /\ \mathit{TimeOut}$$

Using the abbreviations `find_first` and `dec_clock`, and the result data type, we define the evaluation of statements as follows:

Definition B.7 (GCL Semantics: Evaluation of Statements)**.**

```
evaluate Skip s ≝ (OK,s)
evaluate (Assign v e) s ≝
  (OK,s with vars := s.vars |+ (v,eval e s))
evaluate (Seq p1 p2) s ≝
  (let
     (r,t) = evaluate p1 s
   in
     if r = OK then evaluate p2 t else (r,t))
evaluate (If es) s ≝
  case find_first es s of
    NONE ⇒ (Abort,s)
  | SOME (e,p1) ⇒ evaluate p1 s
evaluate (While es) s ≝
  case find_first es s of
    NONE ⇒ (OK,s)
  | SOME (e,p1) ⇒
    if s.clock = 0 then (TimeOut,s)
    else
      (let
         (r,t) = evaluate p1 (dec_clock s)
       in
         if r = OK then evaluate (While es) t else (r,t))
```

For this inductive definition, we have to provide *HOL4* with a termination proof because the internal termination prover does not succeed. As a termination argument, we need to provide a well-founded relation R on `Stmt × state`, i.e., the argument type of `evaluate`. We additionally need to show that each argument of the calls of `evaluate` are R-related to the arguments of the caused recursive calls. This can be done by providing the relation

```
inv_image (measure I LEX measure Stmt_size)
  (λ(xs,s). (s.clock,xs)),
```

i.e., a lexicocraphical composition of measure-based relations on the size of the provided statement and on the state's clock value, as at least one of both decreases with every call.

Given the terminating evaluation of a statement from an initial clock value, we prove that the evaluation terminates in the same resulting state from any greater clock value, and that the clock increase propages through:

Theorem B.3 (Increasing the Clock when Evaluating Statements)**.**

```
⊢ evaluate p s = (OK,t) ⇒
  evaluate p (s with clock := s.clock + ck) =
  (OK,t with clock := t.clock + ck)
```

B.2 Specifications and Weakest Preconditions

As discussed in Section 3.1, it is advantageous to require the clock-freeness of preconditions and postconditions in Hoare specifications. Similar to Definition 3.2, we define:

Definition B.8 (Clockfree Conditions).

$$\textit{clkfree}\ P \overset{\text{def}}{=} \forall s\ k.\ P\ s \Rightarrow P\ (s\ \textit{with clock} := k)$$

Based on the evaluation function, we define a Hoare specification as a precondition and a postcondition in which the precondition establishes both the postcondition and the termination, i.e., a specification of total correctness.

Definition B.9 (Hoare Specification).

$$\begin{array}{l}
\textit{hoare}\ P\ \textit{prog}\ Q \overset{\text{def}}{=} \\
\quad \textit{clkfree}\ P \wedge \textit{clkfree}\ Q\ \wedge \\
\quad \forall s.\ P\ s \Rightarrow \\
\qquad \exists k.\ (\mathbf{let} \\
\qquad\qquad (r,t) = \textit{evaluate prog}\ (s\ \textit{with clock} := k) \\
\qquad\quad \mathbf{in} \\
\qquad\qquad r = \textit{OK} \wedge Q\ t)
\end{array}$$

Similarly, we define the weakest precondition (WP) of a statement:

Definition B.10 (Weakest Precondition).

$$\begin{array}{l}
\textit{wp prog}\ Q\ s \overset{\text{def}}{=} \\
\quad \textit{clkfree}\ Q\ \wedge \\
\quad \exists k.\ (\mathbf{let} \\
\qquad\quad (r,t) = \textit{evaluate prog}\ (s\ \textit{with clock} := k) \\
\qquad \mathbf{in} \\
\qquad\quad r = \textit{OK} \wedge Q\ t)
\end{array}$$

As done in Theorems 3.2 and D.18, we show that a WP is a clockfree condition and that a WP has its eponymous properties:

Theorem B.4 (Clock-Freeness of Weakest Precondition).

$$\vdash \textit{clkfree}\ (\textit{wp prog}\ Q)$$

Theorem B.5 (Properties of the Weakest Precondition).

$$\begin{array}{l}
\vdash \textit{clkfree}\ P \wedge \textit{clkfree}\ Q \Rightarrow \\
\quad \textit{hoare}\ (\textit{wp prog}\ Q)\ \textit{prog}\ Q\ \wedge \\
\quad (\textit{hoare}\ P\ \textit{prog}\ Q \iff \forall s.\ P\ s \Rightarrow \textit{wp prog}\ Q\ s)
\end{array}$$

As it is a useful lemma, we prove that the WP is a monotonic operator.

Theorem B.6 (WP is a Monotonic Operator).

$$\begin{array}{l}
\vdash \textit{clkfree}\ A \wedge \textit{clkfree}\ B \Rightarrow \\
\quad (\forall s.\ A\ s \Rightarrow B\ s) \Rightarrow \\
\quad \forall s.\ \textit{wp}\ p\ A\ s \Rightarrow \textit{wp}\ p\ B\ s
\end{array}$$

Based on these definitions and theorems, we prove the well-known WPs of `Skip`, `Assign`, and `Seq`.

Theorem B.7 (Weakest Precondition of `Skip`).

$$\vdash \textit{clkfree}\ Q \Rightarrow (\textit{wp Skip}\ Q\ s \iff Q\ s)$$

Theorem B.8 (Weakest Precondition of `Assign`).

$$\begin{array}{l}
\vdash \textit{clkfree}\ Q \Rightarrow \\
\quad (\textit{wp}\ (\textit{Assign}\ v\ e)\ Q\ s \iff \\
\quad\ \ Q\ (s\ \textit{with vars} := s.\textit{vars}\ |\!+ (v, \textit{eval}\ e\ s)))
\end{array}$$

Theorem B.9 (Weakest Precondition of `Seq`).

$$\vdash \textit{clkfree}\ Q \Rightarrow (\textit{wp}\ (\textit{Seq}\ p_1\ p_2)\ Q\ s \iff \textit{wp}\ p_1\ (\textit{wp}\ p_2\ Q)\ s)$$

For the selection statement `If`, the adapted semantics, i.e., deterministically selecting the first matching guard, causes the WP to be weaker than usual:

Theorem B.10 (Weakest Precondition of `If`)**.**

$$\begin{array}{l} \vdash \texttt{clkfree}\ Q \Rightarrow \\ \quad (\texttt{wp}\ (\texttt{If}\ es)\ Q\ s \iff \\ \quad\ \exists e.\ \texttt{find\_first}\ es\ s = \texttt{SOME}\ e \wedge \texttt{wp}\ (\texttt{SND}\ e)\ Q\ s) \end{array}$$

As shown in Theorem 3.5, we cannot provide an expressive notation of the WP of `While`, requiring us to prove the corresponding refinement rule independently.

B.3 Contracts and Refinement

We define the language of contracts to be used for refinement as follows:

Definition B.11 (GCL Contracts)**.**

$$\begin{array}{l} \texttt{Contract} = \\ \qquad \texttt{HoareC}\ (\texttt{state} \to \texttt{bool})\ (\texttt{state} \to \texttt{bool}) \\ \quad \mid \texttt{SeqC}\ \texttt{Contract}\ \texttt{Contract} \\ \quad \mid \texttt{IfC}\ ((\texttt{Expr} \times \texttt{Contract})\ \texttt{list}) \\ \quad \mid \texttt{WhileC}\ (\texttt{state} \to \texttt{bool})\ (\texttt{state} \to \texttt{num}) \\ \quad ((\texttt{Expr} \times \texttt{Contract})\ \texttt{list}) \\ \quad \mid \texttt{ProgC}\ \texttt{Stmt} \end{array}$$

To enforce the termination of loops, we use a loop variant as defined in Definition 3.6. In *HOL4*, we define for our calculus:

Definition B.12 (GCL Loop Variant)**.**

$$\begin{array}{l} \texttt{is\_variant}\ i\ v\ es \overset{\text{def}}{=} \\ \quad \texttt{EVERY}\ (\lambda p.\ \forall s.\ i\ s \Rightarrow v\ (\texttt{SND}\ (\texttt{evaluate}\ p\ s)) < v\ s) \\ \qquad (\texttt{MAP SND}\ es) \wedge \forall s\ k.\ v\ s = v\ (s\ \texttt{with}\ \texttt{clock} := k) \end{array}$$

Using this definition, we define the satisfication relation between contracts and statements. For the cases omitted in the following definition, the reader may assume non-satisfication.

Definition B.13 (Satsification of GCL Contracts)**.**

$$\begin{array}{l} \texttt{sat}\ (\texttt{HoareC}\ P\ Q)\ prog \overset{\text{def}}{=} \texttt{hoare}\ P\ prog\ Q \\ \texttt{sat}\ (\texttt{SeqC}\ c_1\ c_2)\ (\texttt{Seq}\ p_1\ p_2) \overset{\text{def}}{=} \texttt{sat}\ c_1\ p_1 \wedge \texttt{sat}\ c_2\ p_2 \\ \texttt{sat}\ (\texttt{IfC}\ ls)\ (\texttt{If}\ rs) \overset{\text{def}}{=} \\ \quad \texttt{MAP FST}\ ls = \texttt{MAP FST}\ rs \wedge \\ \quad \texttt{LIST\_REL}\ (\lambda c\ p.\ \texttt{sat}\ (\texttt{SND}\ c)\ (\texttt{SND}\ p))\ ls\ rs \\ \texttt{sat}\ (\texttt{WhileC}\ i\ v\ ls)\ (\texttt{While}\ rs) \overset{\text{def}}{=} \\ \quad \texttt{MAP FST}\ ls = \texttt{MAP FST}\ rs \wedge \texttt{is\_variant}\ i\ v\ rs \wedge \\ \quad \texttt{LIST\_REL}\ (\lambda c\ p.\ \texttt{sat}\ (\texttt{SND}\ c)\ (\texttt{SND}\ p))\ ls\ rs \\ \texttt{sat}\ (\texttt{ProgC}\ l)\ r \overset{\text{def}}{=} l = r \\ \texttt{sat}\ (\texttt{SeqC}\ v_6\ v_7)\ \texttt{Skip} \overset{\text{def}}{=} \texttt{F} \end{array}$$

As done in Definition 3.8, we define refinement as a relation of contracts:

Definition B.14 (Refinement of GCL Contracts)**.**

$$\texttt{refine}\ c_1\ c_2 \overset{\text{def}}{=} \forall p.\ \texttt{sat}\ c_2\ p \Rightarrow \texttt{sat}\ c_1\ p$$

The refinement relation is reflexive and transitive, as proven in Theorem 2.1. Additionally, the composition of contracts is monotonic with respect to the refinement relation:

Theorem B.11 (Monotonicity of GCL Contract Composition)**.**

```
⊢ (∀a b c.
      refine a b ⇒
      refine (SeqC a c) (SeqC b c) ∧
      refine (SeqC c a) (SeqC c b)) ∧
   (∀a b es e n.
      refine a b ∧ n < LENGTH es ∧ EL n es = (e,a) ⇒
      refine (IfC es) (IfC (LUPDATE (e,b) n es))) ∧
   ∀a b es e n i v.
     refine a b ∧ n < LENGTH es ∧ EL n es = (e,a) ⇒
     refine (WhileC i v es)
       (WhileC i v (LUPDATE (e,b) n es))
```

B.4 Refinement Rules

We continue by proving the correctness of the well-known CbC refinement rules for GCL as presented by Kourie and Watson [KW12]. In addition to the *top-down* refinement rules they introduce, i.e., rules that introduce new statements from Hoare specifications, we prove *bottom-down* refinement rules which allow us to make the mostly syntactic step from contracts back to statements.

Top-Down Refinement Rules

For the top-down refinement rules, we prove the well-known refinement rules. Each rule introduces one statement type, requiring the clock-freeness of the involved conditions and statement-specific side conditions.

For introducing the `Skip` statement, one needs to show that the precondition implies the postcondition:

Theorem B.12 (Refinement Rule: `Skip`)**.**

```
⊢ clkfree P ∧ clkfree Q ∧ (∀s. P s ⇒ Q s) ⇒
  refine (HoareC P Q) (ProgC Skip)
```

For introducing the `Assign` statement, one needs to show that the precondition implies the postcondition after substituting the new value for the assigned variable:

Theorem B.13 (Refinement Rule: `Assign`)**.**

```
⊢ clkfree P ∧ clkfree Q ∧
  (∀s. P s ⇒ Q (s with vars := s.vars |+ (v,eval e s))) ⇒
  refine (HoareC P Q) (ProgC (Assign v e))
```

For introducing the `Seq` statement, one needs to show the clock-freeness of the intermediate condition, requiring the sub-statements to establish the intermediate condition from the precondition, and the postcondition from the intermediate condition:

Theorem B.14 (Refinement Rule: `Seq`)**.**

```
⊢ clkfree P ∧ clkfree M ∧ clkfree Q ⇒
  refine (HoareC P Q) (SeqC (HoareC P M) (HoareC M Q))
```

For introducing the `If` statement, one needs to show that there is a matching guard, requiring each sub-statement to establish the postcondition from the precondition and the guard evaluating to *true*. This well-known rule establishes stronger conditions as necessary for our adapted semantics, as the weakest precondition would only require the first matching guard to have a sub-statement establishing the postcondition. However, for consistency with existing research, we decided to present the well-known refinement rule.

Theorem B.15 (Refinement Rule: `If`).

```
⊢ clkfree P ∧ clkfree Q ∧
  (∀s. P s ⇒ EXISTS (λe. eval e s > 0) es) ⇒
  refine (HoareC P Q)
    (IfC
       (MAP (λx. (x,HoareC (λs. P s ∧ eval x s > 0) Q)) es))
```

For introducing the `While` statement, one needs to show the clock-freeness of the invariant, that the precondition implies the invariant, and the invariant and all guards evaluating to *false* implies the postcondition, requiring each sub-statement to establish the invariant from the invariant and the guard evaluating to *true*. The termination of the loop is enforced separately through the corresponding bottom-up refinement rule.

Theorem B.16 (Refinement Rule: `While`).

```
⊢ clkfree P ∧ clkfree Q ∧ clkfree i ∧ (∀s. P s ⇒ i s) ∧
  (∀s. i s ∧ EVERY (λe. eval e s ≤ 0) es ⇒ Q s) ⇒
  refine (HoareC P Q)
    (WhileC i v
       (MAP (λx. (x,HoareC (λs. i s ∧ eval x s > 0) i)) es))
```

Bottom-Up Refinement Rules

We end this chapter with the bottom-up refinement rules for the composite contracts which allow their refinement back into `ProgC` contracts.

The bottom-up refinement rule for `Seq` is trivial:

Theorem B.17 (Bottom-Up Refinement Rule: `Seq`).

```
⊢ refine (SeqC (ProgC l) (ProgC r)) (ProgC (Seq l r))
```

For `If` and `While`, we introduce the following abbreviation to extract statement from `ProgC` contracts. As it is a partial function, it has the arbitrary value `ARB` as a possible return value.

Definition B.15 (Getting a Statement from a `ProgC`).

```
get_prog cs ≝
  MAP
    (λ(e,c).
        case c of
          HoareC v v1 ⇒ ARB
        | SeqC v4 v5 ⇒ ARB
        | IfC v8 ⇒ ARB
        | WhileC v10 v11 v12 ⇒ ARB
        | ProgC p ⇒ (e,p)) cs
```

From here, the bottom-up refinement rule for `If` is also trivial:

Theorem B.18 (Bottom-Up Refinement Rule: `If`).

```
⊢ EVERY (λ(e,c). ∃p. c = ProgC p) ec ⇒
  refine (IfC ec) (ProgC (If (get_prog ec)))
```

The bottom-up refinement rule for `While` requires to prove the termination of the loop through the variant. As discussed in Section 4.3.2, this imposes challenges for the automatic application of the refinement rule. However, a different design was not feasible due to the limited completion time of this thesis.

Theorem B.19 (Bottom-Up Refinement Rule: `While`).

```
⊢ EVERY (λ (e,c). ∃p. c = ProgC p) ec ∧
  is_variant i v (get_prog ec) ⇒
  refine (WhileC i v ec) (ProgC (While (get_prog ec)))
```

C. The Pancake Semantics

In this thesis, we use a subset of Pancake [Poh+23] presented in Section 2.2.

The semantics of Pancake is formalized as *functional big-step semantics* [Owe+16]. The record data type representing the program state is shown in Figure 2.4, the evaluation function and result types for our language subset are shown below. All definitions occurring below are unchanged from the definitions made in the *CakeML* repository[1], Git commit `721c4576e`.

```
α result =
    Error
  | TimeOut
  | Break
  | Continue
  | Return (α v)
  | Exception mlstring (α v)
evaluate (Skip,s) =def (NONE,s)
evaluate (Dec v shape e prog,s) =def
  case eval s e of
    NONE ⇒ (SOME Error,s)
  | SOME value ⇒
    (let
       (r,t) =
         evaluate
           (prog,s with locals := s.locals |+ (v,value))
     in
       (r,
        t with
        locals := res_var t.locals (v,FLOOKUP s.locals v)))
```

[1]https://github.com/CakeML/cakeml

```
evaluate (Assign Local $v$ $src$,$s$) $\overset{\text{def}}{=}$
  case eval $s$ $src$ of
    NONE ⇒ (SOME Error,$s$)
  | SOME $value$ ⇒
    if is_valid_value $s$.locals $v$ $value$ then
      (NONE,$s$ with locals := $s$.locals |+ ($v$,$value$))
    else (SOME Error,$s$)
evaluate (Assign Global $v$ $src$,$s$) $\overset{\text{def}}{=}$
  case eval $s$ $src$ of
    NONE ⇒ (SOME Error,$s$)
  | SOME $value$ ⇒
    if is_valid_value $s$.globals $v$ $value$ then
      (NONE,$s$ with globals := $s$.globals |+ ($v$,$value$))
    else (SOME Error,$s$)
evaluate (Store $dst$ $src$,$s$) $\overset{\text{def}}{=}$
  case (eval $s$ $dst$,eval $s$ $src$) of
    (NONE,$v_3$) ⇒ (SOME Error,$s$)
  | (SOME (Val $v_8$),NONE) ⇒ (SOME Error,$s$)
  | (SOME (ValWord $addr$),SOME $value$) ⇒
    (case
       mem_stores $addr$ (flatten $value$) $s$.memaddrs $s$.memory
     of
       NONE ⇒ (SOME Error,$s$)
     | SOME $m$ ⇒ (NONE,$s$ with memory := $m$))
  | (SOME (Struct $v_9$),$v_3$) ⇒ (SOME Error,$s$)
evaluate (Store32 $dst$ $src$,$s$) $\overset{\text{def}}{=}$
  case (eval $s$ $dst$,eval $s$ $src$) of
    (NONE,$v_3$) ⇒ (SOME Error,$s$)
  | (SOME (Val $v_8$),NONE) ⇒ (SOME Error,$s$)
  | (SOME (ValWord $adr$),SOME (ValWord $w$)) ⇒
    (case
       mem_store_32 $s$.memory $s$.memaddrs $s$.be $adr$ (w2w $w$)
     of
       NONE ⇒ (SOME Error,$s$)
     | SOME $m$ ⇒ (NONE,$s$ with memory := $m$))
  | (SOME (Val $v_8$),SOME (Struct $v_{15}$)) ⇒ (SOME Error,$s$)
  | (SOME (Struct $v_9$),$v_3$) ⇒ (SOME Error,$s$)
evaluate (StoreByte $dst$ $src$,$s$) $\overset{\text{def}}{=}$
  case (eval $s$ $dst$,eval $s$ $src$) of
    (NONE,$v_3$) ⇒ (SOME Error,$s$)
  | (SOME (Val $v_8$),NONE) ⇒ (SOME Error,$s$)
  | (SOME (ValWord $adr$),SOME (ValWord $w$)) ⇒
    (case
       mem_store_byte $s$.memory $s$.memaddrs $s$.be $adr$ (w2w $w$)
     of
       NONE ⇒ (SOME Error,$s$)
     | SOME $m$ ⇒ (NONE,$s$ with memory := $m$))
  | (SOME (Val $v_8$),SOME (Struct $v_{15}$)) ⇒ (SOME Error,$s$)
  | (SOME (Struct $v_9$),$v_3$) ⇒ (SOME Error,$s$)
```

```
evaluate (Seq $c_1$ $c_2$,$s$) $\overset{\text{def}}{=}$
  (let
     ($r$,$s_1$) = evaluate ($c_1$,$s$)
   in
     if $r$ = NONE then evaluate ($c_2$,$s_1$) else ($r$,$s_1$))
evaluate (If $e$ $c_1$ $c_2$,$s$) $\overset{\text{def}}{=}$
  case eval $s$ $e$ of
    NONE ⇒ (SOME Error,$s$)
  | SOME (ValWord $w$) ⇒
    evaluate (if $w \neq$ 0w then $c_1$ else $c_2$,$s$)
  | SOME (Struct $v_5$) ⇒ (SOME Error,$s$)
evaluate (Break,$s$) $\overset{\text{def}}{=}$ (SOME Break,$s$)
evaluate (Continue,$s$) $\overset{\text{def}}{=}$ (SOME Continue,$s$)
evaluate (While $e$ $c$,$s$) $\overset{\text{def}}{=}$
  case eval $s$ $e$ of
    NONE ⇒ (SOME Error,$s$)
  | SOME (ValWord $w$) ⇒
    if $w \neq$ 0w then
      if $s$.clock = 0 then (SOME TimeOut,empty_locals $s$)
      else
        (let
           ($r$,$s_1$) = evaluate ($c$,dec_clock $s$)
         in
           case $r$ of
             NONE ⇒ evaluate (While $e$ $c$,$s_1$)
           | SOME Error ⇒ ($r$,$s_1$)
           | SOME TimeOut ⇒ ($r$,$s_1$)
           | SOME Break ⇒ (NONE,$s_1$)
           | SOME Continue ⇒ evaluate (While $e$ $c$,$s_1$)
           | SOME (Return $v_5$) ⇒ ($r$,$s_1$)
           | SOME (Exception $v_6$ $v_7$) ⇒ ($r$,$s_1$))
    else (NONE,$s$)
  | SOME (Struct $v_9$) ⇒ (SOME Error,$s$)
evaluate (Return $e$,$s$) $\overset{\text{def}}{=}$
  case eval $s$ $e$ of
    NONE ⇒ (SOME Error,$s$)
  | SOME $value$ ⇒
    if size_of_shape (shape_of $value$) $\leq$ 32 then
      (SOME (Return $value$),empty_locals $s$)
    else (SOME Error,$s$)
evaluate (Raise $eid$ $e$,$s$) $\overset{\text{def}}{=}$
  case (FLOOKUP $s$.eshapes $eid$,eval $s$ $e$) of
    (NONE,$v_3$) ⇒ (SOME Error,$s$)
  | (SOME $sh$,NONE) ⇒ (SOME Error,$s$)
  | (SOME $sh$,SOME $value$) ⇒
    if
      shape_of $value$ = $sh$ $\wedge$
      size_of_shape (shape_of $value$) $\leq$ 32
    then
      (SOME (Exception $eid$ $value$),empty_locals $s$)
    else (SOME Error,$s$)
evaluate (Annot $v_0$ $v_1$,$s$) $\overset{\text{def}}{=}$ (NONE,$s$)
```

D. CbC for Pancake in HOL4

In this chapter, we list the *HOL4* definitions and theorems for our CbC calculus for Pancake. In Section D.1, we begin by defining the clock-freeness und variable-freeness of conditions, as well as composition rules for clockfree conditions. We continue by defining conditions on program states, as well as condition operators in Section D.2. In Section D.3, we define Hoare specifications and WPs for Pancake, followed by the theorems specifying the WP for the single Pancake statements. In Section D.4, we define contracts, their satisfication, and their refinement, and provide theorems specifying the refinement rules of our CbC calculus. Finally, we provide the *HOL4* implementations of our automated refinement proof tactics in Section D.5.

As all proofs within this chapter are developed in *HOL4*, we omit the detailled proof scripts, only providing the proven theorems.

D.1 Clock-Freeness and Variable-Freeness

In this section, we provide the definitions of clock-freeness and variable-freeness of conditions. As our case study in Chapter 4 shows, clock-freeness proofs are common when applying our calculus. Thus, we aim to reduce the proof load by providing theorems about the compositionality of clockfree conditions.

Definitions

We define clock-freeness and variable-freeness as follows:

Definition D.1 (Clock-Freeness of Preconditions).

$$
\begin{aligned}
&\texttt{clkfree\_p}\ P \overset{\text{def}}{=} \\
&\quad \forall s\ k_1\ k_2\,. \\
&\qquad P\ (s\ \texttt{with clock := } k_1) \iff P\ (s\ \texttt{with clock := } k_2)
\end{aligned}
$$

Definition D.2 (Clock-Freeness of Postconditions).

$$
\begin{aligned}
&\texttt{clkfree\_q}\ Q \overset{\text{def}}{=} \\
&\quad \forall r\ s\ k_1\ k_2\,. \\
&\qquad Q\ (r,s\ \texttt{with clock := } k_1) \iff Q\ (r,s\ \texttt{with clock := } k_2)
\end{aligned}
$$

Definition D.3 (Variable-Freeness of Preconditions)**.**

```
varfree_p v P ≝
  ∀s val.
    P s ⇒
    P (s with locals := s.locals |+ (v,val)) ∧
    P (s with locals := s.locals \\ v)
```

Definition D.4 (Variable-Freeness of Postconditions)**.**

```
varfree_q v Q ≝
  ∀r t val.
    Q (r,t) ⇒
    Q (r,t with locals := t.locals |+ (v,val)) ∧
    Q (r,t with locals := t.locals \\ v)
```

Theorems about Compositionality

For the compositionality of clockfree conditions, we begin by proving the trivial base cases, i.e., the clock-freeness of $\lambda s.$ `T`, $\lambda s.$ `F`, and postconditions which only have requirements on the result type.

Theorem D.1 (Clock-Freeness of Preconditions: Base Cases)**.**

```
⊢ clkfree_p (λs. T) ∧ clkfree_p (λs. F)
```

Theorem D.2 (Clock-Freeness of Postconditions: Base Cases)**.**

```
⊢ clkfree_q (λs. T) ∧ clkfree_q (λs. F) ∧
  ∀R. clkfree_q (λ(r,t). R r)
```

We continue by showing that clock-freeness is compositional with the boolean operators for conjunction (∧) and disjunction (∨):

Theorem D.3 (Clock-Freeness of Preconditions: Conjunctions)**.**

```
⊢ clkfree_p P ∧ clkfree_p R ⇒ clkfree_p (λs. P s ∧ R s)
```

Theorem D.4 (Clock-Freeness of Preconditions: Disjunctions)**.**

```
⊢ clkfree_p P ∧ clkfree_p R ⇒ clkfree_p (λs. P s ∨ R s)
```

Theorem D.5 (Clock-Freeness of Postconditions: Conjunctions)**.**

```
⊢ clkfree_q Q ∧ clkfree_q R ⇒
  clkfree_q (λ(r,t). Q (r,t) ∧ R (r,t))
```

Theorem D.6 (Clock-Freeness of Postconditions: Disjunctions)**.**

```
⊢ clkfree_q Q ∧ clkfree_q R ⇒
  clkfree_q (λ(r,t). Q (r,t) ∨ R (r,t))
```

Additionally, we show that clock-freeness is a monotonic operator:

Theorem D.7 (Clock-Freeness of Preconditions: Monotonicity)**.**

```
⊢ (∀s. P s ⟺ P′ s) ⇒ (clkfree_p P ⟺ clkfree_p P′)
```

Theorem D.8 (Clock-Freeness of Postconditions: Monotonicity)**.**

```
⊢ (∀r. Q r ⟺ Q′ r) ⇒ (clkfree_q Q ⟺ clkfree_q Q′)
```

Finally, we prove some selected composition rules that are necessary to deal with the conditions occurring in our refinement rules:

Theorem D.9 (Clock-Freeness: Postconditions from Preconditions)**.**

$\vdash$ *clkfree_p* $P \Rightarrow$
clkfree_q $(\lambda\,(r,t).\ P\ t) \land$
$\forall R.$ *clkfree_q* $(\lambda\,(r,t).\ R\ r \land P\ t)$

Theorem D.10 (Clock-Freeness: Preconditions from Postconditions)**.**

$\vdash$ *clkfree_q* $Q \Rightarrow$ *clkfree_p* $(\lambda s.\ Q\ (r,s))$

Theorem D.11 (Clock-Freeness: Postconditions with Result Requirement)**.**

$\vdash$ *clkfree_q* $Q \Rightarrow$ *clkfree_q* $(\lambda\,(r,t).\ r \neq r' \land Q\ (r,t))$

Theorem D.12 (Clock-Freeness: Postconditions with Result Cases)**.**

$\vdash$ *clkfree_q* $Q \land$ *clkfree_p* $M \Rightarrow$
clkfree_q $(\lambda\,(r,t).$ **if** $r \neq r'$ **then** $Q\ (r,t)$ **else** $M\ t)$

In combination with the clock-freeness of the conditions from the following section, these theorems can be used to prove the clock-freeness of most conditions arising when using our calculus.

D.2 Conditions and Operators

To improve the readability of our refinement rules and to simplify the refinement proofs, we introduce a set of conditions and condition operators, each paired with its clock-freeness theorem. For these clock-freeness theorems, we provide one initial example, as the remaining theorems are structured similarly. Where it is useful, we provide additional theorems about the conditions.

Initially, we prove a monotonicity theorem for constructing postconditions from preconditions:

Theorem D.13 (Monotonicity of Postconditions from Preconditions)**.**

$\vdash (\forall s.\ A\ s \Rightarrow B\ s) \Rightarrow$
$\forall s.\ (\lambda\,(r,t).\ r = r' \land A\ t)\ s \Rightarrow (\lambda\,(r,t).\ r = r' \land B\ t)\ s$

Conditions: Evaluation of Expressions

We begin by defining conditions about the evaluation of expressions, including the requirement of evaluation to a given shape or value, to a word, to a *true* value, and to a *false* value. For the latter two, we prove a law of excluded middle (LEM) and the contradiction between both.

Definition D.5 (Condition: `evaluates`)**.**

evaluates $e\ s \overset{\text{def}}{=} \exists v.$ *eval* $s\ e$ = *SOME* v

Theorem D.14 (Clock-Freeness: `evaluates`)**.**

$\vdash$ *clkfree_p* $(\lambda s.$ *evaluates* $e\ s) \land$ *clkfree_p* (*evaluates* e)

Definition D.6 (Condition: `evaluates_shape`)**.**

evaluates_shape $e\ sh\ s \overset{\text{def}}{=}$
$\exists v.$ *eval* $s\ e$ = *SOME* $v \land$ *shape_of* $v = sh$

Definition D.7 (Condition: `evaluates_to`)**.**

evaluates_to $e\ v\ s \overset{\text{def}}{=}$ *eval* $s\ e$ = *SOME* v

Definition D.8 (Condition: *evaluates_word*)**.**

evaluates_to_word $e\ s \overset{\text{def}}{=} \exists w.$ *eval* $s\ e$ = *SOME* (*ValWord* w)

Definition D.9 (Condition: `evaluates_to_true`).

```
evaluates_to_true e s ≝
  ∃w. eval s e = SOME (ValWord w) ∧ w ≠ 0w
```

Definition D.10 (Condition: `evaluates_to_false`).

```
evaluates_to_false e s ≝
  ∃w. eval s e = SOME (ValWord w) ∧ w = 0w
```

Theorem D.15 (Boolean Conditions: LEM).

```
⊢ evaluates_to_word e s ⇒
    evaluates_to_true e s ∨ evaluates_to_false e s
```

Theorem D.16 (Boolean Conditions: Contradictions).

```
⊢ (evaluates_to_true e s ⇒ ¬evaluates_to_false e s) ∧
    (evaluates_to_false e s ⇒ ¬evaluates_to_true e s)
```

Conditions and Operators: Variables

We continue with the definition of conditions about the value of variables, including variables being equal to constant values and values on the heap. Furthermore, we define a condition and a condition operator for variable substitution.

Definition D.11 (Condition: *var_eq_val_def*).

```
var_eq_val k v val s ≝
  case k of
    Local ⇒ FLOOKUP s.locals v = SOME val
  | Global ⇒ FLOOKUP s.globals v = SOME val
```

Definition D.12 (Condition: *var_eq_mem_def*).

```
var_eq_mem k v ad sh s ≝
  ∃addr value.
    eval s ad = SOME (ValWord addr) ∧
    mem_load sh addr s.memaddrs s.memory = SOME value ∧
    case k of
      Local ⇒ FLOOKUP s.locals v = SOME value
    | Global ⇒ FLOOKUP s.globals v = SOME value
```

Definition D.13 (Condition: *valid_value_def*).

```
valid_value k v e s ≝
  ∃value.
    eval s e = SOME value ∧
    case k of
      Local ⇒ is_valid_value s.locals v value
    | Global ⇒ is_valid_value s.globals v value
```

Definition D.14 (Condition Operator: *subst_def*).

```
subst k v e P s ≝
  ∃value.
    eval s e = SOME value ∧
    case k of
      Local ⇒ P (s with locals := s.locals |+ (v,value))
    | Global ⇒ P (s with globals := s.globals |+ (v,value))
```

Conditions: Exception Shapes

Next, we define a condition requiring the existence of a given exception shape:

Definition D.15 (Condition: *has_eshape_def*).

```
has_eshape eid sh s ≝ FLOOKUP s.eshapes eid = SOME sh
```

Conditions and Operators: Heap

Finally, we define conditions and condition operators about the heap, including conditions for the existence of addresses on the heap and condition operators for memory substitutions:

Definition D.16 (Condition: *addr_in_mem_def*).

```
addr_in_mem a v s ≝
  ∃m. mem_stores a (flatten v) s.memaddrs s.memory = SOME m
```

Definition D.17 (Condition Operator: *mem_subst_def*).

```
mem_subst a v P s ≝
  ∃m. mem_stores a (flatten v) s.memaddrs s.memory = SOME m ∧
      P (s with memory := m)
```

Definition D.18 (Condition: *addr_in_mem_32_def*).

```
addr_in_mem_32 a v s ≝
  ∃m. mem_store_32 s.memory s.memaddrs s.be a (w2w v) =
      SOME m
```

Definition D.19 (Condition Operator: *mem_subst_32_def*).

```
mem_subst_32 a v P s ≝
  ∃m. mem_store_32 s.memory s.memaddrs s.be a (w2w v) =
      SOME m ∧ P (s with memory := m)
```

Definition D.20 (Condition: *addr_in_mem_byte_def*).

```
addr_in_mem_byte a v s ≝
  ∃m. mem_store_byte s.memory s.memaddrs s.be a (w2w v) =
      SOME m
```

Definition D.21 (Condition Operator: *mem_subst_byte_def*).

```
mem_subst_byte a v P s ≝
  ∃m. mem_store_byte s.memory s.memaddrs s.be a (w2w v) =
      SOME m ∧ P (s with memory := m)
```

D.3 Specifications and Weakest Preconditions

To formalize the starting point for our calculus, we define Hoare specifications. Additionally, we prove WPs of Pancake statements as a helpful intermediate representation of the language's semantics.

Definitions and Properties

We begin by defining Hoare specifications and WPs and by proving useful lemmas about their properties:

Definition D.22 (Hoare Specification).

$$\begin{array}{l} \textit{hoare}\ P\ \textit{prog}\ Q \overset{\text{def}}{=} \\ \quad \texttt{clkfree\_p}\ P \land \texttt{clkfree\_q}\ Q \land \\ \quad \forall s.\ P\ s \Rightarrow \\ \qquad \exists k.\ (\textbf{let} \\ \qquad\qquad (r,t) = \texttt{evaluate}\ (\textit{prog},s\ \texttt{with}\ \textit{clock} := k) \\ \qquad\quad \textbf{in} \\ \qquad\qquad r \neq \textit{SOME Error} \land r \neq \textit{SOME TimeOut} \land Q\ (r,t)) \end{array}$$

Theorem D.17 (Hoare Specification: Monotonicity in the Precondition).

$$\vdash (\forall s.\ P\ s \iff P'\ s) \Rightarrow (\texttt{hoare}\ P\ \textit{prog}\ Q \iff \texttt{hoare}\ P'\ \textit{prog}\ Q)$$

Definition D.23 (Weakest Precondition (WP)).

$$\begin{array}{l} \textit{wp}\ \textit{prog}\ Q\ s \overset{\text{def}}{=} \\ \quad \texttt{clkfree\_q}\ Q \land \\ \quad \exists k.\ (\textbf{let} \\ \qquad\quad (r,t) = \texttt{evaluate}\ (\textit{prog},s\ \texttt{with}\ \textit{clock} := k) \\ \qquad \textbf{in} \\ \qquad\quad r \neq \textit{SOME Error} \land r \neq \textit{SOME TimeOut} \land Q\ (r,t)) \end{array}$$

Theorem D.18 (WP: Clock-Freeness).

$$\vdash \texttt{clkfree\_p}\ (\textit{wp}\ \textit{prog}\ Q)$$

Theorem D.19 (WP: Compositionality with $(\land)$).

$$\begin{array}{l} \vdash \texttt{clkfree\_q}\ A \land \texttt{clkfree\_q}\ B \Rightarrow \\ \quad \forall s.\ \textit{wp}\ p\ (\lambda(r,t).\ A\ (r,t) \land B\ (r,t))\ s \iff \\ \qquad \textit{wp}\ p\ A\ s \land \textit{wp}\ p\ B\ s \end{array}$$

Theorem D.20 (WP: Compositionality with $(\lor)$).

$$\begin{array}{l} \vdash \texttt{clkfree\_q}\ A \land \texttt{clkfree\_q}\ B \Rightarrow \\ \quad \forall s.\ \textit{wp}\ p\ (\lambda(r,t).\ A\ (r,t) \lor B\ (r,t))\ s \iff \\ \qquad \textit{wp}\ p\ A\ s \lor \textit{wp}\ p\ B\ s \end{array}$$

Theorem D.21 (WP: Monotonicity).

$$\begin{array}{l} \vdash \texttt{clkfree\_q}\ A \land \texttt{clkfree\_q}\ B \Rightarrow \\ \quad (\forall s.\ A\ s \Rightarrow B\ s) \Rightarrow \\ \quad \forall s.\ \textit{wp}\ \textit{prog}\ A\ s \Rightarrow \textit{wp}\ \textit{prog}\ B\ s \end{array}$$

Theorem D.22 (WP: Compositionality with Result Cases).

$$\begin{array}{l} \vdash \texttt{clkfree\_q}\ Q \land \texttt{clkfree\_p}\ M \Rightarrow \\ \quad \forall s.\ \textit{wp}\ p\ (\lambda(r,t).\ \textbf{if}\ r \neq r'\ \textbf{then}\ Q\ (r,t)\ \textbf{else}\ M\ t)\ s \iff \\ \qquad \textit{wp}\ p\ (\lambda(r,t).\ r \neq r' \land Q\ (r,t))\ s \lor \\ \qquad \textit{wp}\ p\ (\lambda(r,t).\ r = r' \land M\ t)\ s \end{array}$$

WPs of Pancake Statements

Using the preceding definitions, we prove the WPs of Pancake statements:

Theorem D.23 (WP: `Skip`).

$$\vdash \texttt{clkfree\_q}\ Q \Rightarrow (\textit{wp}\ \textit{Skip}\ Q\ s \iff Q\ (\textit{NONE},s))$$

Theorem D.24 (WP: `Dec`).

$$\begin{array}{l} \vdash \texttt{clkfree\_q}\ Q \Rightarrow \\ \quad (\textit{wp}\ (\textit{Dec}\ v\ \textit{sh}\ \textit{src}\ \textit{prog})\ Q\ s \iff \\ \quad\ \texttt{evaluates}\ \textit{src}\ s \land \\ \quad\ \texttt{subst}\ \textit{Local}\ v\ \textit{src}\ (\textit{wp}\ \textit{prog}\ (\texttt{reset\_subst}\ v\ s\ Q))\ s) \end{array}$$

Theorem D.25 (WP: `Assign`).

$$\begin{array}{l}
\vdash \mathtt{clkfree\_q}\ Q \Rightarrow \\
\quad (\mathtt{wp}\ (\mathtt{Assign}\ k\ v\ src)\ Q\ s \iff \\
\quad\ \ \mathtt{valid\_value}\ k\ v\ src\ s \land \\
\quad\ \ \mathtt{subst}\ k\ v\ src\ (\lambda s.\ Q\ (\mathtt{NONE},s))\ s)
\end{array}$$

Theorem D.26 (WP: `Store`).

$$\begin{array}{l}
\vdash \mathtt{clkfree\_q}\ Q \Rightarrow \\
\quad (\mathtt{wp}\ (\mathtt{Store}\ dest\ src)\ Q\ s \iff \\
\quad\ \ \exists\, addr\ val. \\
\quad\quad \mathtt{evaluates\_to}\ dest\ (\mathtt{ValWord}\ addr)\ s \land \\
\quad\quad \mathtt{evaluates\_to}\ src\ val\ s \land \mathtt{addr\_in\_mem}\ addr\ val\ s \land \\
\quad\quad \mathtt{mem\_subst}\ addr\ val\ (\lambda s.\ Q\ (\mathtt{NONE},s))\ s)
\end{array}$$

Theorem D.27 (WP: `Store32`).

$$\begin{array}{l}
\vdash \mathtt{clkfree\_q}\ Q \Rightarrow \\
\quad (\mathtt{wp}\ (\mathtt{Store32}\ dest\ src)\ Q\ s \iff \\
\quad\ \ \exists\, addr\ val. \\
\quad\quad \mathtt{evaluates\_to}\ dest\ (\mathtt{ValWord}\ addr)\ s \land \\
\quad\quad \mathtt{evaluates\_to}\ src\ (\mathtt{ValWord}\ val)\ s \land \\
\quad\quad \mathtt{addr\_in\_mem\_32}\ addr\ val\ s \land \\
\quad\quad \mathtt{mem\_subst\_32}\ addr\ val\ (\lambda s.\ Q\ (\mathtt{NONE},s))\ s)
\end{array}$$

Theorem D.28 (WP: `StoreByte`).

$$\begin{array}{l}
\vdash \mathtt{clkfree\_q}\ Q \Rightarrow \\
\quad (\mathtt{wp}\ (\mathtt{StoreByte}\ dest\ src)\ Q\ s \iff \\
\quad\ \ \exists\, addr\ val. \\
\quad\quad \mathtt{evaluates\_to}\ dest\ (\mathtt{ValWord}\ addr)\ s \land \\
\quad\quad \mathtt{evaluates\_to}\ src\ (\mathtt{ValWord}\ val)\ s \land \\
\quad\quad \mathtt{addr\_in\_mem\_byte}\ addr\ val\ s \land \\
\quad\quad \mathtt{mem\_subst\_byte}\ addr\ val\ (\lambda s.\ Q\ (\mathtt{NONE},s))\ s)
\end{array}$$

Theorem D.29 (WP: `Seq`).

$$\begin{array}{l}
\vdash \mathtt{clkfree\_q}\ Q \Rightarrow \\
\quad (\mathtt{wp}\ (\mathtt{Seq}\ p_1\ p_2)\ Q\ s \iff \\
\quad\ \ \mathtt{wp}\ p_1\ (\lambda\,(r,t).\ r = \mathtt{NONE} \land \mathtt{wp}\ p_2\ Q\ t)\ s \lor \\
\quad\ \ \mathtt{wp}\ p_1\ (\lambda\,(r,t).\ r \neq \mathtt{NONE} \land Q\ (r,t))\ s)
\end{array}$$

Theorem D.30 (WP: `If`).

$$\begin{array}{l}
\vdash \mathtt{clkfree\_q}\ Q \Rightarrow \\
\quad (\mathtt{wp}\ (\mathtt{If}\ e\ c_1\ c_2)\ Q\ s \iff \\
\quad\ \ \mathtt{evaluates\_to\_word}\ e\ s \land \\
\quad\ \ (\mathtt{evaluates\_to\_true}\ e\ s \Rightarrow \mathtt{wp}\ c_1\ Q\ s) \land \\
\quad\ \ (\mathtt{evaluates\_to\_false}\ e\ s \Rightarrow \mathtt{wp}\ c_2\ Q\ s))
\end{array}$$

Theorem D.31 (WP: `Break`).

$$\vdash \mathtt{clkfree\_q}\ Q \Rightarrow (\mathtt{wp}\ \mathtt{Break}\ Q\ s \iff Q\ (\mathtt{SOME}\ \mathtt{Break},s))$$

Theorem D.32 (WP: `Continue`).

$$\vdash \mathtt{clkfree\_q}\ Q \Rightarrow (\mathtt{wp}\ \mathtt{Continue}\ Q\ s \iff Q\ (\mathtt{SOME}\ \mathtt{Continue},s))$$

Theorem D.33 (WP: `Raise`).

$$\begin{array}{l} \vdash \texttt{clkfree\_q}\ Q \Rightarrow \\ \quad (\texttt{wp}\ (\texttt{Raise}\ eid\ e)\ Q\ s \iff \\ \quad\ \exists sh\ val. \\ \qquad \texttt{has\_eshape}\ eid\ sh\ s \wedge \texttt{evaluates\_to}\ e\ val\ s \wedge \\ \qquad \texttt{shape\_of}\ val = sh \wedge \texttt{size\_of\_shape}\ sh \leq 32 \wedge \\ \qquad Q\ (\texttt{SOME}\ (\texttt{Exception}\ eid\ val),\texttt{empty\_locals}\ s)) \end{array}$$

Theorem D.34 (WP: `Return`).

$$\begin{array}{l} \vdash \texttt{clkfree\_q}\ Q \Rightarrow \\ \quad (\texttt{wp}\ (\texttt{Return}\ e)\ Q\ s \iff \\ \quad\ \exists val. \\ \qquad \texttt{evaluates\_to}\ e\ val\ s \wedge \\ \qquad \texttt{size\_of\_shape}\ (\texttt{shape\_of}\ val) \leq 32 \wedge \\ \qquad Q\ (\texttt{SOME}\ (\texttt{Return}\ val),\texttt{empty\_locals}\ s)) \end{array}$$

Theorem D.35 (WP: `Annot`).

$$\vdash \texttt{clkfree\_q}\ Q \Rightarrow (\texttt{wp}\ (\texttt{Annot}\ a\ b)\ Q\ s \iff Q\ (\texttt{NONE},s))$$

D.4 Refinement and Refinement Rules

In this section, we define contracts, their satisfication, and their refinement, and prove some of the properties of these definitions. Using these definitions and theorems, we prove the refinement rules for our CbC calculus for Pancake.

Definitions and Properties

We begin by defining the loop variant as a clock-independent measure on program states which is required to decrease every loop iteration:

Definition D.24 (Loop Variant).

$$\begin{array}{l} \texttt{is\_variant}\ i\ v\ p \overset{\text{def}}{=} \\ \quad (\forall s.\ i\ s \Rightarrow v\ (\texttt{SND}\ (\texttt{evaluate}\ (p,s))) < v\ s) \wedge \\ \quad \forall s\ k_1\ k_2.\ v\ (s\ \texttt{with clock := } k_1) = v\ (s\ \texttt{with clock := } k_2) \end{array}$$

Next, we define contracts and their satisfication. We provide one type of contract per compositional statement type, in addition to a contract for Hoare specifications and a *don't care* contract. For all cases not displayed in the definition of the satisfication, the reader may assume non-satisfication.

Definition D.25 (Contract).

$$\begin{array}{l} \alpha\ \texttt{Contract} = \\ \quad\ \ \texttt{HoareC}\ (\alpha\ \texttt{state} \to \texttt{bool})\ (\alpha\ \texttt{result option} \times \alpha\ \texttt{state} \to \texttt{bool}) \\ \quad |\ \texttt{DecC mlstring shape}\ (\alpha\ \texttt{exp})\ (\alpha\ \texttt{Contract}) \\ \quad |\ \texttt{SeqC}\ (\alpha\ \texttt{Contract})\ (\alpha\ \texttt{Contract}) \\ \quad |\ \texttt{IfC}\ (\alpha\ \texttt{exp})\ (\alpha\ \texttt{Contract})\ (\alpha\ \texttt{Contract}) \\ \quad |\ \texttt{WhileC}\ (\alpha\ \texttt{exp})\ (\alpha\ \texttt{state} \to \texttt{bool})\ (\alpha\ \texttt{state} \to \texttt{num})\ (\alpha\ \texttt{Contract}) \\ \quad |\ \texttt{PanC}\ (\alpha\ \texttt{prog}) \\ \quad |\ \texttt{DCC} \end{array}$$

Definition D.26 (Satisfication of Contracts)**.**

$$\begin{array}{l}
\texttt{sat (HoareC } P\ Q\texttt{)}\ prog \overset{\text{def}}{=} \texttt{hoare}\ P\ prog\ Q \\
\texttt{sat (DecC}\ nl\ sl\ el\ c\texttt{) (Dec}\ nr\ sr\ er\ p\texttt{)} \overset{\text{def}}{=} \\
\quad nl = nr \wedge sl = sr \wedge el = er \wedge \texttt{sat}\ c\ p \\
\texttt{sat (SeqC}\ c_1\ c_2\texttt{) (Seq}\ p_1\ p_2\texttt{)} \overset{\text{def}}{=} \texttt{sat}\ c_1\ p_1 \wedge \texttt{sat}\ c_2\ p_2 \\
\texttt{sat (IfC}\ l\ c_1\ c_2\texttt{) (If}\ r\ p_1\ p_2\texttt{)} \overset{\text{def}}{=} l = r \wedge \texttt{sat}\ c_1\ p_1 \wedge \texttt{sat}\ c_2\ p_2 \\
\texttt{sat (WhileC}\ l\ i\ v\ c\texttt{) (While}\ r\ p\texttt{)} \overset{\text{def}}{=} \\
\quad l = r \wedge \texttt{sat}\ c\ p \wedge \texttt{is\_variant}\ i\ v\ p \\
\texttt{sat (PanC}\ l\texttt{)}\ r \overset{\text{def}}{=} l = r \\
\texttt{sat DCC}\ v_0 \overset{\text{def}}{=} T
\end{array}$$

Finally, we define the refinement relation between contracts, and prove that it is reflexive and transitive, and that the composition of contracts is monotonic with respect to the refinement relation:

Definition D.27 (Refinement)**.**

$$\texttt{refine}\ c_1\ c_2 \overset{\text{def}}{=} \forall prog.\ \texttt{sat}\ c_2\ prog \Rightarrow \texttt{sat}\ c_1\ prog$$

Theorem D.36 (Refinement: Reflexivity and Transitivity)**.**

$$\begin{array}{l}
\vdash \texttt{refine}\ A\ A \\
\vdash \texttt{refine}\ A\ B \wedge \texttt{refine}\ B\ C \Rightarrow \texttt{refine}\ A\ C
\end{array}$$

Theorem D.37 (Contract Composition and Refinement: Monotonicity)**.**

$$\begin{array}{l}
\vdash \texttt{refine}\ A\ B \Rightarrow \texttt{refine (DecC}\ v\ sh\ exp\ A\texttt{) (DecC}\ v\ sh\ exp\ B\texttt{)} \\
\vdash \texttt{refine}\ A\ B \Rightarrow \\
\quad \texttt{refine (SeqC}\ A\ C\texttt{) (SeqC}\ B\ C\texttt{)} \wedge \\
\quad \texttt{refine (SeqC}\ C\ A\texttt{) (SeqC}\ C\ B\texttt{)} \\
\vdash \texttt{refine}\ A\ B \Rightarrow \\
\quad \texttt{refine (IfC}\ e\ A\ C\texttt{) (IfC}\ e\ B\ C\texttt{)} \wedge \\
\quad \texttt{refine (IfC}\ e\ C\ A\texttt{) (IfC}\ e\ C\ B\texttt{)} \\
\vdash \texttt{refine}\ A\ B \Rightarrow \texttt{refine (WhileC}\ e\ i\ v\ A\texttt{) (WhileC}\ e\ i\ v\ B\texttt{)}
\end{array}$$

Top-Down Refinement Rules

Using the preceding definitions, we prove the *top-down* refinement rules of our CbC calculus, i.e., refinement rules that introduce new contract or statement types from Hoare specifications, or refinement rules that modify the conditions on a Hoare specification.

We begin with the refinement rules for strengthening a postcondition or weakening a precondition on a Hoare specification.

Theorem D.38 (Refinement Rule: Strengthen Postcondition)**.**

$$\begin{array}{l}
\vdash \texttt{clkfree\_q}\ Q \wedge (\forall s.\ Q'\ s \Rightarrow Q\ s) \Rightarrow \\
\quad \texttt{refine (HoareC}\ P\ Q\texttt{) (HoareC}\ P\ Q'\texttt{)}
\end{array}$$

Theorem D.39 (Refinement Rule: Weaken Postcondition)**.**

$$\begin{array}{l}
\vdash \texttt{clkfree\_p}\ P \wedge (\forall s.\ P\ s \Rightarrow P'\ s) \Rightarrow \\
\quad \texttt{refine (HoareC}\ P\ Q\texttt{) (HoareC}\ P'\ Q\texttt{)}
\end{array}$$

We continue with the refinement rules which introduce new contract or statement types. For each statement type of our language subset, we provide at least one refinement rule. For `Dec` and `Seq`, we provide multiple refinement rules to target different situations, i.e., declaring a variable from a constant or from memory, and using a sequence that terminates deterministically in the first sub-program, the second sub-program, or possibly in both.

Theorem D.40 (Refinement Rule: `Skip`).

```
⊢ clkfree_p P ∧ clkfree_q Q ∧ (∀s. P s ⇒ Q (NONE,s)) ⇒
  refine (HoareC P Q) (PanC Skip)
```

Theorem D.41 (Refinement Rule: `Dec` (Fresh Variable, Constant)).

```
⊢ clkfree_p P ∧ clkfree_q Q ∧ varfree_p v P ∧ varfree_q v Q ∧
  (∀s. P s ⇒ evaluates_to src val s) ⇒
  refine (HoareC P Q)
    (DecC v sh src
       (HoareC (λs. P s ∧ var_eq_val Local v val s) Q))
```

Theorem D.42 (Refinement Rule: `Dec` (Fresh Variable, Memory)).

```
⊢ clkfree_p P ∧ clkfree_q Q ∧ varfree_p v P ∧ varfree_q v Q ∧
  (∀s. P s ⇒ evaluates_shape (Load sh ad) sh s) ∧
  ¬MEM v (var_exp ad) ⇒
  refine (HoareC P Q)
    (DecC v sh (Load sh ad)
       (HoareC (λs. P s ∧ var_eq_mem Local v ad sh s) Q))
```

Theorem D.43 (Refinement Rule: `Assign`).

```
⊢ clkfree_p P ∧ clkfree_q Q ∧
  (∀s. P s ⇒
       valid_value k v src s ∧
       subst k v src (λs. Q (NONE,s)) s) ⇒
  refine (HoareC P Q) (PanC (Assign k v src))
```

Theorem D.44 (Refinement Rule: `Store`).

```
⊢ clkfree_p P ∧ clkfree_q Q ∧
  (∀s. P s ⇒
       ∃addr val.
         evaluates_to dest (ValWord addr) s ∧
         evaluates_to src val s ∧ addr_in_mem addr val s ∧
         mem_subst addr val (λs. Q (NONE,s)) s) ⇒
  refine (HoareC P Q) (PanC (Store dest src))
```

Theorem D.45 (Refinement Rule: `Store32`).

```
⊢ clkfree_p P ∧ clkfree_q Q ∧
  (∀s. P s ⇒
       ∃addr val.
         evaluates_to dest (ValWord addr) s ∧
         evaluates_to src (ValWord val) s ∧
         addr_in_mem_32 addr val s ∧
         mem_subst_32 addr val (λs. Q (NONE,s)) s) ⇒
  refine (HoareC P Q) (PanC (Store32 dest src))
```

Theorem D.46 (Refinement Rule: `StoreByte`).

```
⊢ clkfree_p P ∧ clkfree_q Q ∧
  (∀s. P s ⇒
       ∃addr val.
         evaluates_to dest (ValWord addr) s ∧
         evaluates_to src (ValWord val) s ∧
         addr_in_mem_byte addr val s ∧
         mem_subst_byte addr val (λs. Q (NONE,s)) s) ⇒
  refine (HoareC P Q) (PanC (StoreByte dest src))
```

Theorem D.47 (Refinement Rule: `Seq` (Terminating in First Sub-Program)).

```
⊢ clkfree_p P ∧ clkfree_q Q ⇒
  refine (HoareC P Q)
    (SeqC (HoareC P (λ(r,t). r ≠ NONE ∧ Q (r,t))) DCC)
```

Theorem D.48 (Refinement Rule: `Seq` (Terminating in Second Sub-Program)).

```
⊢ clkfree_p P ∧ clkfree_q Q ∧ clkfree_p M ⇒
  refine (HoareC P Q)
    (SeqC (HoareC P (λ(r,t). r = NONE ∧ M t)) (HoareC M Q))
```

Theorem D.49 (Refinement Rule: `Seq` (Terminating in Any Sub-Program)).

```
⊢ clkfree_p P ∧ clkfree_q Q ∧ clkfree_p M ⇒
  refine (HoareC P Q)
    (SeqC
       (HoareC P
          (λ(r,t). if r ≠ NONE then Q (r,t) else M t))
       (HoareC M Q))
```

Theorem D.50 (Refinement Rule: `If`).

```
⊢ clkfree_p P ∧ clkfree_q Q ∧
  (∀s. P s ⇒ evaluates_to_word e s) ⇒
  refine (HoareC P Q)
    (IfC e (HoareC (λs. P s ∧ evaluates_to_true e s) Q)
       (HoareC (λs. P s ∧ evaluates_to_false e s) Q))
```

Theorem D.51 (Refinement Rule: `Break`).

```
⊢ clkfree_p P ∧ clkfree_q Q ∧ (∀s. P s ⇒ Q (SOME Break,s)) ⇒
  refine (HoareC P Q) (PanC Break)
```

Theorem D.52 (Refinement Rule: `Continue`).

```
⊢ clkfree_p P ∧ clkfree_q Q ∧ (∀s. P s ⇒ Q (SOME Continue,s)) ⇒
  refine (HoareC P Q) (PanC Continue)
```

For the refinement rule for `While`, we introduce two abbreviations for the precondition and postcondition used within the rule:

Definition D.28 (Precondition for `While` Refinement Rule).

```
while_body_pre i e ≝ (λs. i s ∧ evaluates_to_true e s)
```

Definition D.29 (Postcondition for `While` Refinement Rule)**.**

```
while_body_post i QB QR QE ≝
  (λ (r,t).
       case r of
         NONE ⇒ i t
       | SOME Error ⇒ i t
       | SOME TimeOut ⇒ i t
       | SOME Break ⇒ QB t
       | SOME Continue ⇒ i t
       | SOME (Return v) ⇒ QR (t,v)
       | SOME (Exception eid e) ⇒ QE (t,eid,e))
```

Theorem D.53 (Refinement Rule: `While`)**.**

```
⊢ clkfree_p P ∧ clkfree_q Q ∧ clkfree_p i ∧ (∀s. P s ⇒ i s) ∧
  (∀s. i s ⇒ evaluates_to_word e s) ∧
  (∀s. i s ∧ evaluates_to_false e s ⇒ Q (NONE,s)) ∧
  (∀t. QB t ⇒ Q (NONE,t)) ∧
  (∀t v. QR (t,v) ⇒ Q (SOME (Return v),t)) ∧
  (∀t eid v. QE (t,eid,v) ⇒ Q (SOME (Exception eid v),t)) ⇒
  refine (HoareC P Q)
    (WhileC e i v
       (HoareC (while_body_pre i e)
          (while_body_post i QB QR QE)))
```

Theorem D.54 (Refinement Rule: `Raise`)**.**

```
⊢ clkfree_p P ∧ clkfree_q Q ∧
  (∀s. P s ⇒
       ∃sh val.
         has_eshape eid sh s ∧ evaluates_to e val s ∧
         shape_of val = sh ∧
         size_of_shape (shape_of val) ≤ 32 ∧
         Q (SOME (Exception eid val),empty_locals s)) ⇒
  refine (HoareC P Q) (PanC (Raise eid e))
```

Theorem D.55 (Refinement Rule: `Return`)**.**

```
⊢ clkfree_p P ∧ clkfree_q Q ∧
  (∀s. P s ⇒
       ∃val.
         evaluates_to e val s ∧
         size_of_shape (shape_of val) ≤ 32 ∧
         Q (SOME (Return val),empty_locals s)) ⇒
  refine (HoareC P Q) (PanC (Return e))
```

Theorem D.56 (Refinement Rule: `Annot`)**.**

```
⊢ clkfree_p P ∧ clkfree_q Q ∧ (∀s. P s ⇒ Q (NONE,s)) ⇒
  refine (HoareC P Q) (PanC (Annot t1 t2))
```

Theorem D.57 (Refinement Rule: `DCC`)**.**

```
⊢ refine DCC (PanC prog)
```

Bottom-Up Refinement Rules

Finally, we prove four *bottom-up* refinement rules which transform compositional contracts containing only `PanC` sub-contracts, i.e., contracts with completed refinement, back into `PanC` contracts, i.e., contracts representing the corresponding program.

These refinement rules are mostly trivial, except the rule for `While`, which requires proving the termination of the loop by proving the variant.

Theorem D.58 (Bottom-Up Refinement Rule: `Dec`)**.**

```
⊢ refine (DecC v sh src (PanC prog))
    (PanC (Dec v sh src prog))
```

Theorem D.59 (Bottom-Up Refinement Rule: `Seq`)**.**

```
⊢ refine (SeqC (PanC l) (PanC r)) (PanC (Seq l r))
```

Theorem D.60 (Bottom-Up Refinement Rule: `If`)**.**

```
⊢ refine (IfC e (PanC l) (PanC r)) (PanC (If e l r))
```

Theorem D.61 (Bottom-Up Refinement Rule: `While`)**.**

```
⊢ is_variant i v p ⇒
  refine (WhileC e i v (PanC p)) (PanC (While e p))
```

D.5 Automating Refinement

In this section, we provide the implementation of our eight automated refinement proof tactics for HOL4 as discussed in Section 4.3. All tactics described in this section use the *simpset* `pan_refinement_ss` which consists of the definitions and theorems provided in this chapter, selected definitions and theorems from the Pancake semantics, and theorems from HOL4's `pred_setTheory`, `finite_mapTheory`, and `wordsTheory`.

The eight refinement tactics all derive from the following tactic, which implements the procedure described in Section 4.3.1, except the use of *blastLib* or *HolSmtLib*. This tactic takes two arguments: (1) a refinement rule to apply, and (2) a set of additional theorems to apply during simplification.

```
val pan_refinement_thms_tac = fn refinement_rule =>
      fn extra_thms =>
  (* apply refinement rule, unabbreviate side condition goals *)
  rw[]
  >> irule refinement_rule
  >> unabbrev_all_tac
  >> gvs[]

  (* apply pan_refinement_ss with rw until unchanged *)
  >> rpt (CHANGED_TAC (rw_tac pan_refinement_ss extra_thms))

  (* try to instantiate existential quantifiers *)
  >> TRY (HINT_EXISTS_TAC)

  (* split into all possible cases and simplify using fs *)
  >> every_case_tac
  >> fs[]

  (* apply pan_refinement_ss with gvs and case splits until unchanged *)
  >> rpt (CHANGED_TAC (global_simp_tac {elimvars = true,
                                        strip = true,
                                      droptrues = true,
                                      oldestfirst = true}
                                     pan_refinement_ss
                                     extra_thms
                     >> every_case_tac))

  (* try to match remaining implication assumptions to the goal *)
  >> TRY (first_x_assum $ irule)

  (* prepare proof by contradiction *)
  >> spose_not_then assume_tac

  (* solve remaining clock-freeness goals using definitions *)
  >> gvs[...];
```

The next three tactics add a last step to the preceding tactic: the application of *blastLib*, i.e., `FULL_BBLAST_TAC`, or *HolSmtLib*, i.e., `z3_tac` or `z3o_tac`. This can be used to solve remaining goals of word arithmetic. As discussed in Section 4.3.1, the user may decide which of these options to use, depending on the size of the goal to prove and the required reliability.

```
val pan_refinement_thms_tac_blast = fn refinement_rule =>
      fn extra_thms =>
  pan_refinement_thms_tac refinement_rule extra_thms
  >> FULL_BBLAST_TAC;

val pan_refinement_thms_tac_z3 = fn refinement_rule =>
      fn extra_thms =>
  pan_refinement_thms_tac refinement_rule extra_thms
  >> z3_tac extra_thms;

val pan_refinement_thms_tac_z3o = fn refinement_rule =>
      fn extra_thms =>
  pan_refinement_thms_tac refinement_rule extra_thms
  >> z3o_tac extra_thms;
```

The remaining four tactics provide a short-hand for the application of any preceding tactic without additional theorems for simplification.

```
val pan_refinement_tac = fn rule =>
  pan_refinement_thms_tac rule [];

val pan_refinement_tac_blast = fn rule =>
  pan_refinement_thms_tac_blast rule [];

val pan_refinement_tac_z3 = fn rule =>
  pan_refinement_thms_tac_z3 rule [];

val pan_refinement_tac_z3o = fn rule =>
  pan_refinement_thms_tac_z3o rule [];
```