Bachelor's Thesis

Verified SMT-based Translation Validation

Author:
Heiko Becker

Supervisors:
Sigurd Schneider
Prof. Dr. Sebastian Hack

Reviewers:
Prof. Dr. Sebastian Hack
Prof. Dr. Gert Smolka

submitted on
Monday 16th March, 2015

Saarland University
Faculty of Natural Sciences and Technology I

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

Date Signature

This thesis develops a translation validation framework that relies on an SMT solver. Translation validation verifies a run of an optimization and not the optimization itself. The framework encodes program semantics of a restricted class of functional first-order programs in SMT formulas. Therefore it is independent of the applied optimization. If the SMT formula for a source and a target program is unsatisfiable, then the target program is a correct implementation of the source program. The formula generation is designed to handle partial functions in an expression language. We prove that the framework validates all optimizations that operate on the semantics of the restricted programs. Since general program equivalence is undecidable, we restrict the programs to contain conditionals, variable assignments and arithmetic expressions, but no loops. The development extends the LVC framework and is carried out in the Coq proof assistant.

Acknowledgements

I would like to thank my advisors Sigurd Schneider and Prof. Sebastian Hack for giving me the oppportunity to write this thesis and for supporting me during the development. I would like to thank Sigurd Schneider for giving me this possibility of deepening my understanding of theorem proving and correctness in all categories. I am grateful to Prof. Sebastian Hack for reviewing my thesis as the first reviewer. I would also like to thank Prof. Gert Smolka for being the second reviewer of this thesis. At last I would like to thank my family and my friends for always supporting me.

Contents

1 Errata

- **09-Avril-2015** In the definition of the conversion, the function calls were still written with variables ($\overline{x}$) although there was supposed to be a list of expressions ($\overline{e}$)
- **09-May-2015** In the section about IL/F there was the lemma *terminates_impl_star*2 from the development missing. Furthermore the lemmata for the prove of the Soundness Theorem Lemma 14 and Lemma 17 have been rewritten to remove some typos and clarify the statement of the lemmas.

2 Introduction

2.1 Description of the Problem

In a verified compiler, each optimization is verified during the development. As stated by Pnueli, Siegel, and Singerman [18], translation validation is a means of decoupling the correctness proof from the optimization. Instead of proving an optimization correct, the compiler verifies correctness of an optimization run at compile-time.
In this thesis we develop a translation validation [18] framework that relies on an SMT solver [19]. The framework decides whether one program is a correct implementation of the other for a restricted class of functional first-order programs. We encode the semantics of two programs in an SMT formula that is unsatisfiable if the program after applying an optimization is a correct implementation of the program before the optimization. Since general program equivalence is undecidable, we restrict the programs such that they contain conditionals, variable assignments and arithmetic expressions, but no loops.
The development extends the LVC framework [20] and is carried out in the Coq proof assistant [25]. The LVC framework describes the IL intermediate language, which we use to express the programs that we validate.

2.2 An Introductory Example

Consider the program

```
let x = i + i in
let y = x * 2 in
f y
```

Listing 1: Source program s

Assume this is the source program s, before applying an optimization. Suppose an optimization produced the target program t:

```
let x = 4 * i in
f x
```

Listing 2: Target program t

We define an encoding of program semantics into one SMT formulas for both programs. The SMT formula must encode the control flow of the programs. We call programs where every name is bound only once, *renamed apart*. Two programs are *renamed apart from each other*, if they are renamed apart and each variable is only bound in either of the programs, but not both. We α-convert Listing 2 such that it is renamed apart from Listing 1:

```
let z = 4 * i in
f z
```

Listing 3: Listing 2 renamed apart from Listing 1

Note that Listing 1 and Listing 3 may share free variables, which we use to express that the programs have the same inputs. In the generated formula every SMT variable corresponds to the IL variable with the same name. For the statement *let* $x = e$ *in* s we add the formula $\mathtt{x} = \mathtt{e}$ with a conjunction to the SMT formula of s.
For conditionals *if* c *then* s_1 *else* s_2, where the subformula for s_1 is $\mathtt{s}_1$ and $\mathtt{s}_2$ for s_2, the formula

simply is $(\texttt{c} \rightarrow \texttt{s}_1) \wedge (\neg \texttt{c} \rightarrow \texttt{s}_2)$.
Function applications $f\,(e_1, \ldots, e_n)$ are encoded as $\texttt{f}\ (\texttt{e}_1, \ldots, \texttt{e}_\texttt{n})$ where `f` is an uninterpreted function symbol. In the SMT logic `f` is a predicate of type:

$$\text{list bitvector} \rightarrow \text{bool}$$

The precise definition of the translation is given in Section 6.
The formula for the source program from Listing 1 is:

$$\begin{aligned} \texttt{x} = \texttt{i} + \texttt{i} \ \wedge \\ \texttt{y} = \texttt{x} * 2 \ \wedge \\ \texttt{f}\ (\texttt{y}) \end{aligned} \tag{1}$$

The formula for Listing 2 is:

$$\begin{aligned} \texttt{z} = 4 * \texttt{i} \ \wedge \\ \texttt{f}\ (\texttt{z}) \end{aligned} \tag{2}$$

SMT solvers decide whether formulas are satisfiable. We need a decision whether the target program is a correct implementation of the source program. We provide a formula that is satisfiable if the two programs compute different results. If the solver finds a valid assignment to the free variables and predicate symbols, t is not a correct implementation of s. Otherwise the solver is unable to find such an assignment. Therefore t is a correct implementation of s. To achieve this, we use two translation functions, one for the translation of the source program and one for the target program. For the source program the formula will be the same as in Equation 1. $[\cdot]^+$ denotes the source translation. For the target program every occurrence of a uninterpreted function symbol is negated. $[\cdot]^-$ denotes the target translation. This leads to the program formulas

$$\begin{aligned} [s]^+ &= \texttt{x} = \texttt{i} + \texttt{i} \wedge \texttt{y} = \texttt{x} * 2 \wedge \texttt{f}\ (\texttt{y}) \\ [t]^- &= \texttt{z} = 4 * \texttt{i} \wedge \neg \texttt{f}\ (\texttt{z}) \end{aligned}$$

The conjunction of the formulas (Equation 3) is unsatisfiable if and only if the programs are equal.

$$\begin{aligned} \texttt{x} = \texttt{i} + \texttt{i} \ \wedge \ \texttt{y} = \texttt{x} * 2 \ \wedge \ \texttt{f}\ (\texttt{y}) \wedge \\ \texttt{z} = 4 * \texttt{i} \ \wedge \ \neg \texttt{f}\ (\texttt{z}) \end{aligned} \tag{3}$$

Note that the only difference is how function applications are translated. This is needed to correlate the solvers decision to program equivalence.
The SMT translation is done in such a way that the formula is unsatisfiable if and only if y and z are equal. Due to the literals $\texttt{f}\ (\texttt{y})$ and $\neg\ \texttt{f}\ (\texttt{z})$ the solver searches for assignments such that the results of the calls differ.
Assume the solver is able to find an assignment that satisfies Equation 3. This means that the constraints in the formula are not sufficient to ensure that `f` is always called with the same value. Hence the programs are semantically different. Assume the solver is unable to find such an assignment. This means that, for all assignments to the free variables, `f` is always called with the same value. Therefore unsatisfiability of the formula is sufficient for t being a correct implementation of s.

2.3 Contribution of the Thesis

This thesis describes an encoding of the semantics of the restricted IL programs into SMT formulas. The translation function also accounts for partial functions in the expression language. The main technical result is a theorem that shows that if the SMT formula for a source program s and a target program t is unsatisfiable, then t is a correct implementation of s.

2.4 Outline

First, we give an overview of the functional interpretation of IL and define the bitvector expression language in Section 3. We define our restricted program subset and show that in our subset it is decidable whether a program terminates or crashes.
Next, we give a general overview of SMT and explain the formalization of SMT in Coq in Section 4. We explain how to integrate an SMT solver in the framework and relate expression evaluation in IL and SMT.
As we want our expression language to be able to express division, Section 5 explains how we must account for partial functions in the SMT formulas.
In Section 6 we properly define the translation from functional IL to SMT formulas and establish the soundness theorem. Section 7 contains the proof of the theorem.
Finally, Section 8 gives an overview of related work and Section 9 explains possible future work.

3 IL

This chapter gives an overview of the IL language in which the programs that we want to validate are expressed. The IL intermediate language has been developed by Schneider [20] in his masters thesis. The languages expressiveness has been improved in [22]. IL has been developed in the proof assistant Coq [25].

3.1 Bitvector Expression Language

As stated in Schneider's thesis, IL has little dependence on the expression language, which is used for expression evaluation. For this thesis, a subset of the bitvector arithmetics used in SMT solvers is implemented and is used as the expression language of IL. Thereby we do not need to prove the equivalence of the expression languages for SMT and IL. As the IL expression language, we formalize 32-bit two's complement bitvector arithmetics. We take 32-bit words as values:

$$\mathbb{V} \;=\; \{x_0 \ldots x_{32} |\; x_i \in \{0,1\}\}$$

We define $\mathcal{V}$ as the set of variable names. Value environments are partial functions from names to values:

$$V : \mathcal{V} \rightarrow \mathbb{V}_\perp$$

Expression evaluation is defined as a partial function that takes an expression and a partial environment and returns a value or crashes:

$$[\![\cdot]\!] : \mathit{Exp} \rightarrow (\mathcal{V} \rightarrow \mathbb{V}_\perp) \rightarrow \mathbb{V}_\perp$$

Evaluation of expressions is a partial function as we wanted to be able to express division of bitvectors, which is undefined for the bitstring 0^{32}.
We define a function from $\mathbb{V}$ to $\{0,1\}$:

$$\mathbb{B}(v) \;=\; \begin{cases} 0 & \text{if } v = 0^{32} \\ 1 & \text{otherwise} \end{cases}$$

Assume $c \in \mathbb{V}$ and $v \in \mathcal{V}$. Figure 1 describes the grammar of the expression language.

$\mathit{Exp} \ni e$	$::= \quad e \;+_b\; e$	Binary Addition
	$\mid e \;-_b\; e$	Binary Substraction
	$\mid e \;*_b\; e$	Binary Multiplication
	$\mid e \;/_b\; e$	Binary Division
	$\mid e \;=_b\; e$	Equality Test
	$\mid e \;\neq_b\; e$	Inequality Test
	$\mid -_b\, e$	Negation
	$\mid c$	Constants
	$\mid v$	Variables

Figure 1: Grammar for the expression language

We define the set of variables of an expression:

$$\mathrm{Vars}(e) \subseteq \mathcal{V}$$

$$\begin{array}{rcll} \mathrm{Vars}(e_1 \,\square\, e_2) & = & \mathrm{Vars}(e_1) \,\cup\, \mathrm{Vars}(e_2) & \square \in \{\, +_b,\, -_b,\, *_b,\, /_b,\, =_b \neq_b \,\} \\ \mathrm{Vars}(-_b\; e_1) & = & \mathrm{Vars}(e_1) & \\ \mathrm{Vars}(c) & = & \emptyset & \\ \mathrm{Vars}(v) & = & \{\, v \,\} & \end{array}$$

Bit operations are based on the book of Parhami [16]. The only function that is not total is division. We define a helper function $div\colon \mathbb{V} \to \mathbb{V} \to \mathbb{V}_\perp$ as follows:

$$\mathrm{div}\; e_1\; e_2 \;=\; \begin{cases} \perp & \text{if } e_2 = 0^{32} \\ 0^{31}1 \;+_b\; \mathrm{div}(e_1 \;-_b\; e_2)\; e_2 & \text{if } e_2 \;\leq_b\; e_1 \\ 0^{32} & \text{otherwise} \end{cases}$$

Then we define the semantics of $/_b$:

$$[\![e_1 \;/_b\; e_2]\!] = \begin{cases} div(-_b\; e_1)\; (-_b\; e_2) & \text{if } e_1 \;\leq_b\; 0^{32} \wedge e_2 \;\leq_b\; 0^{32} \\ -_b\; div(e_1)\; (-_b\; e_2) & \text{if } e_1 \;\geq_b\; 0^{32} \wedge e_2 \;\leq_b\; 0^{32} \\ -_b\; div(-_b\; e_1)\; (e_2) & \text{if } e_1 \;\leq_b\; 0^{32} \wedge e_2 \;\geq_b\; 0^{32} \\ div(e_1)\; (e_2) & \text{if } e_1 \;\geq_b\; 0^{32} \wedge e_2 \;\geq_b\; 0^{32} \end{cases}$$

We lift $[\![\cdot]\!]$ to lists:

$$[\![e_1, \ldots, e_n]\!]\, V \;=\; \begin{cases} \perp & \text{if } [\![e_1]\!]\, V \;=\; \perp \\ \perp & \text{if } [\![e_2, \ldots, e_n]\!]\, V \;=\; \perp \\ v_1, \ldots, v_n & \text{otherwise} \end{cases}$$

We do not show any relation between the implemented bitvector arithmetics and arithmetics on integers, but take a definition of $[\![\cdot]\!]$ as specification.

3.2 Syntax and Semantic of IL

We give an overview of the Syntax and Semantics of IL. The overview is based on the definitions from Schneider, Smolka, and Hack [22]. Assume e ranges over the expression language defined in subsection 3.1. Let $\mathcal{L}$ denote the set of function names. Assume $f \in \mathcal{L}$ and $x \in \mathcal{V}$. Figure 2 shows the grammar of the used subset of IL.

$$\begin{array}{rll} \mathit{Term} \ni s, t ::= & \mathit{let}\; x \;=\; e\; \mathit{in}\; s & \text{variable binding} \\ \mid & \mathit{if}\; e\; \mathit{then}\; s\; \mathit{else}\; t & \text{conditional} \\ \mid & e & \text{value} \\ \mid & f\,(\overline{e}) & \text{application} \end{array}$$

Figure 2: Grammar of the IL subset we use [22]

To translate the programs into SMT formulas, we must α-convert them. Therefore we use the functional semantics of IL (IL/F) to express the programs that we want to validate. General program equivalence is undecidable. Therefore programs are restricted in such a way that they only contain variable assignments, conditionals, value statements, and function calls. As function definitions are disallowed, program evaluation stops at function calls and value statements. The programs are loop free.
We disallow external function calls. In subsection 9.1 we outline how external function calls could be integrated into the framework.
Next we give the formal definitions of our subset of IL/F according to Schneider, Smolka, and Hack [22]. A function environment L is defined as a partial function:

$$L : \mathcal{L} \rightarrow \ \mathit{closure}_{\perp}$$

A closure (V, $\overline{x}$, s) is a triple consisting of a value environment V , a list of parameter names $\overline{x}$ and the function body s. An IL/F state is defined as triple consisting of a function environment, a value environment and an IL/F statement:

$$\sigma : (\mathcal{L} \rightarrow \ \mathit{closure}_{\perp}) \times (\mathcal{V} \rightarrow \ \mathbb{V}_{\perp}) \times \mathit{Term}$$

The transition relation for IL/F states from our subset is defined in Figure 3.

$$\textsc{Op}\ \frac{[\![e]\!]\, V \ = \ v}{(L \mid V \mid \mathit{let}\ x \ = \ e\ \mathit{in}\ s) \xrightarrow{\tau} (L \mid V\,[x \mapsto v] \mid s)}$$

$$\textsc{Cond}\ \frac{[\![e]\!]\, V \ = \ v \qquad \mathbb{B}\ (\mathrm{v}) = \mathrm{i}}{(L \mid V \mid \mathit{if}\ v\ \mathit{then}\ s_0\ \mathit{else}\ s_1) \xrightarrow{\tau} (L \mid V \mid s_i)}$$

$$\textsc{App}\ \frac{f \notin L' \qquad [\![\overline{e}]\!]\, V \ = \ \overline{v}}{(L; f : (V', \overline{x}, s); L' \mid V \mid f\,(\overline{e})) \xrightarrow{\tau} (L; f : (V', \overline{x}, s) \mid V'\,[\overline{x} \mapsto \overline{v}] \mid s)}$$

Figure 3: Transition relation [22]

3.3 Implementation Relation of Programs in IL

In Schneider, Smolka, and Hack [22], section 3, the implementation relation $\lesssim$ for IL states used in this thesis is defined. Assume $(L \mid V \mid s) \lesssim (L \mid V \mid t)$ holds. Then:

- whenever s terminates with a value v, t must terminate with v too
- whenever s diverges, t diverges too
- whenever s crashes, no restrictions on t are imposed

3.4 Program Evaluation in the IL/F Subset

IL/F is restricted to tail calls. Note that by disallowing function definitions, each evaluation step will reduce to a subterm of the program. Thereby evaluation stops at either an application or a value. We call function applications, denoted as $f\,(\overline{e})$, and value statements, denoted as e, leafs.

We define an inductive relation that describes when a program evaluates until such a leaf, and call this *leaf-evaluation*:

$$\text{T-Ret}\ \frac{[\![e]\!]\, V\ =\ v}{(L \mid V \mid e)\ \Downarrow^t\ (L \mid V \mid e)}$$

$$\text{T-Goto}\ \frac{[\![\overline{e}]\!]\, V\ =\ \overline{v}}{(L \mid V \mid f\,(\overline{e}))\ \Downarrow^t\ (L \mid V \mid f\,(\overline{e}))}$$

$$\text{T-Step}\ \frac{(L \mid V \mid s) \xrightarrow{a} \sigma\text{'} \qquad \sigma\text{'}\ \Downarrow^t\ \sigma\text{"} \qquad \forall\ f\,(\overline{x}),\ \mathrm{s} \neq f\,(\overline{x})}{\sigma\ \Downarrow^t\ \sigma\text{"}}$$

Figure 4: Rules for $\Downarrow^t$

T-Ret is the first base case for leaf-evaluation. A value statement e leaf-evaluates if and only if the value of e exists and can be computed for a given value environment V.
T-Goto is the second base case. A function call $f\,(\overline{e})$ leaf-evaluates if and only if the value of $\bar{e}$ exists and can be computed for a given value environment V. Note that we ignore whether the function is defined in the label environment L or not.
T-Step describes how we can extend leaf-evaluation by arbitrary states. Assume for two states σ and σ', there exists a transition in the IL/F semantics $\sigma \xrightarrow{a} \sigma$'. If we can prove that $\sigma\text{'} \Downarrow^t \sigma\text{"}$, and σ is not a function call, we can conclude that $\sigma \Downarrow^t \sigma\text{"}$. With the last restriction, the rule T-Step disallows transitions under App. This guarantees that no loops can be entered.
Next we define an inductive relation for *leaf-crashing* states analogously:

$$\text{C-Goto}\ \frac{[\![\overline{e}]\!]\, V\ =\ \bot}{(L \mid V \mid f\,(\overline{e}))\ \Downarrow^c\ (L \mid V \mid f\,(\overline{e}))}$$

$$\text{C-Base}\ \frac{\neg\ \exists \sigma,\ (L \mid V \mid s)\ \xrightarrow{a}\ \sigma \qquad \forall\ \mathrm{f}\ \overline{x},\ \mathrm{s} \neq f\,(\overline{x})}{(L \mid V \mid s)\ \Downarrow^c\ (L \mid V \mid s)}$$

$$\text{C-Step}\ \frac{(L \mid V \mid s) \xrightarrow{a} \sigma \qquad \sigma\ \Downarrow^c\ \sigma\text{'} \qquad \forall\ \mathrm{f}\ \overline{x},\ \mathrm{s} \neq f\,(\overline{x})}{(L \mid V \mid s)\ \Downarrow^c\ \sigma\text{'}}$$

Figure 5: Rules for $\Downarrow^c$

C-Goto is the first base case for crashing program evaluations. A function call crashes if and only if the value of $\bar{e}$ does not exist and cannot be computed for the given value environment V.
C-Base is the second base case. For any state $(L \mid V \mid s)$, if there is no transition possible according to the IL/F semantics the evaluation has crashed in $(L \mid V \mid s)$. Therefore we also treat $(L \mid V \mid s)$ as a base case for leaf-crashing.
C-Step encodes how we can extend a crashing evaluation by arbitrary states. Assume for two states σ and σ', there exists a transition in the IL/F semantics $\sigma \xrightarrow{a} \sigma$'. If we can prove that $\sigma\text{'} \Downarrow^c \sigma\text{"}$, and σ does not contain a function call, we can conclude that $\sigma \Downarrow^c \sigma\text{"}$. The last restriction is applied for the same reason as for leaf-termination.

Lemma 1
Let $(L \mid V \mid s)$ be an IL/F state and let s be renamed apart. If $(L \mid V \mid s) \Downarrow^t (L' \mid V' \mid s')$ or $(L \mid V \mid s) \Downarrow^c (L' \mid V' \mid s')$ holds then V and V' agree on the values of the free variables of s.

Proof.
The proof is by induction over the structure of $\Downarrow^t$, respectively $\Downarrow^c$.
Each case closes with the fact that we renamed apart s. Hence no variable binding can overwrite another.

□

Next we show a lemma that relates leaf-evaluation to evaluation in the IL/F subset.

Lemma 2
Let $(L \mid V \mid s)$ be an IL/F state. Whenever $(L \mid V \mid s) \Downarrow^t (L \mid V' \mid s')$ holds, then

$$(L \mid V \mid s) \xrightarrow{\tau} (L \mid V \mid s') \wedge (\exists f\ \overline{e}.\ s' = f(\overline{e})) \vee (\exists e.\ s' = e)$$

Proof.
The prove is by induction over the structure of $\Downarrow^t$. The base cases follow from the defintion of $\Downarrow^t$. The inductive step closes by the same argument with the inductive hypothesis. □

3.5 Leaf-evaluation or Leaf-crashing is Decidable

As our subset of IL is loop free, we prove that for each program in our subset it is decidable whether it leaf-evaluates or leaf-crashes.

Lemma 3
Let s be an IL/F program that contains no function definitions or external function calls. Let V be a value environment. Then:

$$\exists V'\, s', \forall L,\ (L \mid V \mid s) \Downarrow^t (L \mid V' \mid s') \vee (L \mid V \mid s) \Downarrow^c (L \mid V' \mid s')$$

Proof.
The proof is by induction over the structure of s. The base cases are value statements and function calls. First assume s is a value statement e. Let

$$[\![e]\!]\, V = v$$

Therefore T-Ret holds. Otherwise the expression does not evaluate and C-Base holds.
Assume s is a function call $f(\overline{e})$. Let

$$[\![\overline{e}]\!]\, V = \overline{v}$$

Then T-Goto holds. Otherwise an expression in $\overline{e}$ does not evaluate. Therefore

$$[\![\overline{e}]\!]\, V = \bot$$

holds. This case finishes by C-Goto.

In the induction step only conditionals and variable definitions may occur. Function definitions or external function calls are not in our subset of IL/F. Let

$$s \equiv \mathit{let}\ x = e\ \mathit{in}\ s'$$

First we distinguish whether the expression e evaluates to a value v. Let

$$[\![e]\!]\, V \;=\; v$$

As the expression evaluates, we know by Op of the IL/F semantics

$$(L \mid V \mid \mathit{let}\ x \;=\; e\ \mathit{in}\ s') \xrightarrow{\tau} (L \mid V\,[x \mapsto v] \mid s')$$

With the inductive hypothesis we know there is s'' and V'' such that either

$$(L \mid V\,[x \mapsto v] \mid s') \Downarrow^{t} (L \mid V'' \mid s'')$$

or

$$(L \mid V\,[x \mapsto v] \mid s') \Downarrow^{c} (L \mid V'' \mid s'')$$

Therefore we can either apply the T-Step or the C-Step rule and finish the proof. Assume that

$$[\![e]\!]\, V \;=\; \bot$$

It is clear that $(L \mid V \mid \mathit{let}\ x \;=\; e\ \mathit{in}\ s)$ cannot be further evaluated. Therefore C-Base holds in this case.

Let

$$s \equiv \mathit{if}\ c\ \mathit{then}\ s_0\ \mathit{else}\ s_1$$

First we distinguish whether the expression c evaluates or not. Let

$$[\![c]\!]\, V \;=\; v \;\wedge\; \mathbb{B}(v) = i$$

Therefore we know that

$$(L \mid V \mid \mathit{if}\ c\ \mathit{then}\ s_0\ \mathit{else}\ s_1) \xrightarrow{\tau} (L \mid V \mid s_i)$$

Then this case finishes by a case distinction over the inductive hypothesis. If there exists no value for c, the case closes with C-Base.

□

3.6 Independence from Function Environments

We prove that the definition of leaf-evaluation $\Downarrow^{t}$ is insensitive to the function environment L. Assume that a statement s, given the function environment L_1 and the value environment V, evaluates until it reaches a leaf. We show that for all environments L_2, there exists L_2', such that s, given the value environment V, evaluates to the same leaf.

Lemma 4
Let $(L_1 \mid V \mid s)$, $(L_2 \mid V \mid s)$ be IL/F states. Assume V' is a value environment and s' an IL/F statement. Then:

$$\begin{aligned} & (L_1 \mid V \mid s) \Downarrow^{t} (L_1' \mid V' \mid s') \\ \Longrightarrow\ & \exists\, L_2',\ (L_2 \mid V \mid s) \Downarrow^{t} (L_2' \mid V' \mid s') \end{aligned}$$

Proof.
The proof is by induction over $\Downarrow^t$.
The base cases for T-Ret and T-Goto trivially hold as the environment cannot change.
In the induction step we conclude from T-Step that there is a state σ such that

$$(L_1 \mid V \mid s) \xrightarrow{a} \sigma$$

and

$$\sigma \Downarrow^t (L_1' \mid V' \mid s')$$

By a case distinction over

$$(L_1 \mid V \mid s) \xrightarrow{a} \sigma$$

we can construct the environment L_2' in each step from the inductive hypothesis. Therefore the function environment cannot change when doing a single step. With this we can close the proof. □

We show an analogous lemma for leaf-crashing programs. Assume that a statement s, given the function environment L_1 and the value environment V, crashes at a certain state $(L_1' \mid V' \mid s')$. We show that for all environments L_2, there exists L_2', such that s, given the value environment V crashes at $(L_2' \mid V' \mid s')$.

Lemma 5
Let $(L_1 \mid V \mid s)$, $(L_2 \mid V \mid s)$ be IL/F states. Assume V' is a value environment and s' an IL/F statement. Then:

$$\begin{aligned} & (L_1 \mid V \mid s) \Downarrow^c (L_1' \mid V' \mid s') \\ \implies & \exists\, L_2',\ (L_2 \mid V \mid s) \Downarrow^c (L_2' \mid V' \mid s') \end{aligned}$$

Proof.
The proof is analogous. As we disallowed function definitions it cannot be possible that the function environment changes during evaluation. □

4 Satisfiability Modulo Theories (SMT)

4.1 Introduction to SMT

In this chapter we give detail about our formalization of SMT in Coq. We explained before that we want to use an SMT solver [6] to verify wether a target program is a correct implementation of the source program. As explained by Barrett et al. [2], SMT solvers do not check general satisfiability for formulas (as SAT solvers [13]), but they check satisfiability modulo a given theory.
For this thesis, we formalize a theory that ranges over quantifier free bit-vector expressions with uninterpreted function symbols and lists [3].
Formulas of the implemented logic must be quantifier free and may contain arithmetic expressions over fixed length bitvectors as well as uninterpreted function symbols. In this development, we use the same sub-language of arithmetic expressions (subsection 3.1) in IL/F and in the SMT language. Thereby we do not need to prove equivalence of expression evaluation in SMT and IL/F. In contrast to the partial value environments of the IL/F semantics, the SMT logic uses total environments[2]. We account for this difference by requiring that the IL/F value environments are defined on the free variables of the programs when necessary.

4.2 Integration of an External Solver

An SMT formula is passed to the solver and the answer of the solver has to be parsed in by some external interface. We have not implemented solver communication in our framework.
An SMT solver may produce three different results when given an input formula: satisfiable, unsatisfiable or timeout. If one trusts the SMT solver, these three results suffice. To validate the results, additional information is required.
If the formula is satisfiable, there exists a valid assignment for the formula. Therefore the solver must provide this assignment. Validation of this assignment is easy.
The opposite answer is that the formula is unsatisfiable. The SMT solver can prove that there is no satisfying assignment. In this case the solver provides a certificate that proves the fact that there cannot be a valid assignment. This certificate must then be parsed and the proof must be validated.
The third one is a timeout, which means that the solver could not determine a solution. In this case we know nothing about the satisfiability of the formula. We have to make a decision which is sound. In our approach we treat a timeout equivalent to satisfiable and conclude that the target program is not a correct implementation of the source program.
This thesis works under the assumption that the solver produces sound answers. It is also possible to do additional validation of the solvers answer. Armand et al. [1] implemented a validator for proofs of certain solvers in Coq. We do not use their approach because the solvers for which they can validate the certificates do not yet support our implemented logic.

4.3 Syntax and Semantics of SMT

To formally argue about the satisfiability of program formulas we detail our formalization of SMT in Coq. For uninterpreted function symbols we use the same set of names $\mathcal{L}$ as for function names in IL/F. Assume that $\texttt{e}_1$, $\texttt{e}_2$ range over the defined expression language and that `f` is an uninterpreted function symbol. Assume $\texttt{b} \in \{\texttt{true}, \texttt{false}\}$. We define the following grammar for SMT statements:

$\mathtt{s} ::= \mathtt{s} \wedge \mathtt{s}$	Conjunction
$\mid \mathtt{s} \vee \mathtt{s}$	Disjunction
$\mid \mathtt{s} \rightarrow \mathtt{s}$	Implication
$\mid \mathtt{e}_1 = \mathtt{e}_2$	Constraint
$\mid \mathtt{ite}\ (\mathtt{e}_1)\ (\mathtt{s})\ (\mathtt{s})$	Conditional
$\mid \neg\, \mathtt{s}$	Negation
$\mid \mathtt{f}\ (\mathtt{e}_1 \ldots \mathtt{e}_n)$	Predicate evaluation
$\mid \mathtt{b}$	Constants

The implemented subset of SMT uses the bitvector arithmetics defined in subsection 3.1 as the expression language. We simplify here by using a wrapper for the evaluation function. Thereby expression evaluation has the same semantics in SMT and IL/F for expressions. In SMT theories, denotation ($[\![\cdot]\!]$) must be total as well as environments and functions.[2] We will denote only total environments using E:

$$E : \mathcal{V} \rightarrow \mathbb{V}$$

We define expression evaluation in SMT as a total function:

$$[\![\cdot]\!]_{\text{SMT}} : \mathit{Exp} \longrightarrow (\mathcal{V} \longrightarrow \mathbb{V}) \longrightarrow \mathbb{V}$$

As evaluation is done under total environments, the only possibility where an expression may not evaluate in the bitvector arithmetics is when an operation is undefined on its operands values. In such cases, SMT solvers are free to define the denotation by returning a value of the solvers choice. To account for this, we fix a value $u \in \mathbb{V}$ and define $[\![\cdot]\!]_{\text{SMT}}$ as follows:

$$[\![e]\!]_{\text{SMT}}\, E = \begin{cases} v & \text{if } [\![e]\!]\, E = v \\ u & \text{otherwise} \end{cases}$$

Note that we do not make any assumptions over u. We lift $[\![\cdot]\!]_{\text{SMT}}$ to lists:

$$[\![e_1, \ldots, e_n]\!]_{\text{SMT}}\, E = [\![e_1]\!]_{\text{SMT}}\, E, \ldots, [\![e_n]\!]_{\text{SMT}}\, E$$

We define a predicate environment as a function that takes a name and an argument list and returns an SMT constant:

$$F : \mathcal{L} \rightarrow \text{list}\, \mathit{Exp} \rightarrow \{\mathtt{true}, \mathtt{false}\}$$

The semantics of the SMT language are defined in Figure 6.

$$\text{M-And}\ \frac{F\ E \models \mathtt{s}_1 \qquad F\ E \models \mathtt{s}_2}{F\ E \models \mathtt{s}_1 \wedge \mathtt{s}_2} \quad \text{M-Or1}\ \frac{F\ E \models \mathtt{s}_1}{F\ E \models \mathtt{s}_1 \vee \mathtt{s}_2} \quad \text{M-Or2}\ \frac{F\ E \models \mathtt{s}_2}{F\ E \models \mathtt{s}_1 \vee \mathtt{s}_2}$$

$$\text{M-True}\ \frac{}{F\ E \models \mathtt{true}} \quad \text{M-Neg}\ \frac{\neg\ F\ E \models \mathtt{s}}{F\ E \models \neg\ \mathtt{s}}$$

$$\text{M-Pred}\ \frac{[\![\overline{e}]\!]\ E\ =\ \overline{v} \qquad \text{F}\ \mathtt{f}\ \overline{v} = \mathtt{true}}{F\ E \models \mathtt{f}\ (\overline{\mathtt{e}})} \quad \text{M-Constr}\ \frac{[\![e_1]\!]_{\text{SMT}}\ E\ =\ [\![e_2]\!]_{\text{SMT}}\ E}{F\ E \models \mathtt{e}_1 = \mathtt{e}_2}$$

$$\text{M-Impl}\ \frac{F\ E \models \mathtt{s}_1 \Longrightarrow F\ E \models \mathtt{s}_2}{F\ E \models \mathtt{s}_1 \longrightarrow \mathtt{s}_2}$$

$$\text{M-Itetrue}\ \frac{[\![c]\!]_{\text{SMT}}\ E\ =\ v \qquad \mathbb{B}(v) \neq 0^{32} \qquad F\ E \models \mathtt{s}_1}{F\ E \models \mathtt{ite}\ (\mathtt{c})\ (\mathtt{s}_1)\ (\mathtt{s}_2)}$$

$$\text{M-Itefalse}\ \frac{[\![c]\!]_{\text{SMT}}\ E\ =\ v \qquad \mathbb{B}(v) = 0^{32} \qquad F\ E \models \mathtt{s}_2}{F\ E \models \mathtt{ite}\ (\mathtt{c})\ (\mathtt{s}_1)\ (\mathtt{s}_2)}$$

Figure 6: Semantics of the SMT language

The `ite` operator is also defined in the book of Barrett et al. [2]. It is semantically equivalent to a conditional. Our SMT language contains it for readability.
For ease of writing we define the following notations:

$$\models\ p := \forall\ F\ E, F\ E \models p$$

$$\not\models\ p := \forall\ F\ E, F\ E \not\models p$$

By $\hat{V}$ we denote the partial IL/F environment V lifted to a total environment:

$$\hat{V} = \begin{cases} v & \text{if } V\,x\ =\ v \\ 0^{32} & \text{otherwise} \end{cases}$$

We define the function $\text{Vars}(\mathtt{s}) \subseteq \mathcal{V}$ for SMT statements:

$$\begin{aligned}
\text{Vars}(\mathtt{s}_1\ \wedge\ \mathtt{s}_2) &= \text{Vars}(\mathtt{s}_1)\ \cup\ \text{Vars}(\mathtt{s}_2) \\
\text{Vars}(\mathtt{s}_1\ \vee\ \mathtt{s}_2) &= \text{Vars}(\mathtt{s}_1)\ \cup\ \text{Vars}(\mathtt{s}_2) \\
\text{Vars}(\mathtt{s}_1\ \rightarrow\ \mathtt{s}_2) &= \text{Vars}(\mathtt{s}_1)\ \cup\ \text{Vars}(\mathtt{s}_2) \\
\text{Vars}(\mathtt{e}_1 = \mathtt{e}_2) &= \text{Vars}(e_1)\ \cup \text{Vars}(e_2) \\
\text{Vars}(\mathtt{ite}\ (\mathtt{c})\ (\mathtt{s}_1)\ (\mathtt{s}_2)) &= \text{Vars}(c)\ \cup\ \text{Vars}(\mathtt{s}_1)\ \cup\ \text{Vars}(\mathtt{s}_2) \\
\text{Vars}(\neg\ \mathtt{s}_1) &= \text{Vars}(s_1) \\
\text{Vars}(\mathtt{f}\ (\overline{\mathtt{e}})) &= \text{Vars}(\overline{e}) \\
\text{Vars}(\mathtt{b}) &= \{\ \}
\end{aligned}$$

Note that it is possible to use an uninterpreted function symbol with argument lists of different length.

State of the art SMT solvers support uninterpreted function symbols taking lists as arguments.[3] We tried small examples with the SMT solver Z3.[6]

4.4 IL/F Evaluation Implies SMT Evaluation

Lemma 6
Let V be an IL/F environment and e be a bitvector expression.
Then:

$$[\![e]\!]\, V \;=\; v \Rightarrow\; [\![e]\!]_{\mathrm{SMT}}\, \hat{V} \;=\; v$$

and

$$[\![\overline{e}]\!]\, V \;=\; \overline{v} \Rightarrow\; [\![\overline{e}]\!]_{\mathrm{SMT}}\, \hat{V} \;=\; \overline{v}$$

Proof.
The proof is by induction over e. The base cases trivially hold by the definition of $[\![\cdot]\!]$. The induction step follows directly from the inductive hypothesis.
Lemma 6 is lifted to expression lists $\overline{e}$ by a straight forward induction on $\overline{e}$.

□

5 Handling Partially Defined Operators

In subsection 3.1 we defined division of bitvectors as a partial function. In Section 4 we explained that SMT evaluation must be total. We account for an undefined result of expression evaluation by returning the value u. In this section we explain why this unknown value u influences satisfiability and how we must account for this. We will need additional constraints in program formulas. We refer to those constraints as *guards*.

5.1 Guards in the Source Transformation

Assume an example program s:

```
let x = y /b z in
f (x *b z)
```

Listing 4: Program with undefined behavior for z = 0^{32}

and its optimized version t:

```
f y
```

Listing 5: Optimized program without undefined behavior

If z is 0^{32}, the source program contains a division by zero, hence it is broken. We want to allow optimizations to remove this computation as $\lesssim$ is designed such that in the case where z is 0^{32}, t is a correct implementation of s. Therefore, the framework must treat t as a correct implementation of s, too.
To get an idea how we must account for the possibility of a division by zero, we look at the SMT formulas for the two programs. As explained in Section 2, the SMT formula for Listing 4 is:

$$[s]^+ = \mathtt{x} = \mathtt{y} /_\mathtt{b} \mathtt{z} \wedge \mathtt{f}\ (\mathtt{x} *_\mathtt{b} \mathtt{z})$$

The SMT formula for Listing 5 is:

$$[t]^- = \neg\, \mathtt{f}\ (\mathtt{y})$$

Obviously the two programs compute the same result for all possible assignments to `y` and `z` except for `z` = 0^{32}. But these two formulas cannot prove equivalence of the two programs.
As explained before, in SMT theories every function must be total.[2] In the formalized expression language the result of a division by zero is the unknown value u. This means that for the mapping

$$\{\mathtt{y} \mapsto u, \mathtt{z} \mapsto 0^{32}\}$$

the value of `x` is u. Thereby, the formula is satisfiable and t is not a correct implementation of s. We need a constraint such that the solver cannot find satisfying assignments where `z` is assigned to 0^{32}. We add the constraint to the formula for Listing 4:

$$\neg\, \mathtt{z} = 0^{32} \wedge \mathtt{x} = \mathtt{y} /_\mathtt{b} \mathtt{z} \wedge \mathtt{f}\ (\mathtt{x} *_\mathtt{b} \mathtt{z})$$

For all assignments to `z` that are different form 0^{32} the constraint does not influence satisfiability of the formula. If the solver tries to construct an assignment where `z` is assigned to 0^{32} the constraint

$$\neg\, \mathtt{z} = 0^{32}$$

is not satisfiable. As we add the constraint as a conjunction the whole formula becomes unsatisfiable in the case where `z` is assigned 0^{32}. Therefore the formula is always unsatisfiable and our framework treats t as a correct implementation of s.

5.2 Guards in the Target Transformation

Assume the constant c to be equivalent to u. Take the example source program s consisting of a single function call to f with the constant c:

```
f  c
```

Listing 6: Optimized program without undefined computation

Assume an optimization produced the target program t:

```
f  (y /b 0^32)
```

Listing 7: Program with possibility of undefined computation

The formula for Listing 6 is:

$$\texttt{f}\ (\texttt{c})$$

The formula for Listing 7 is:

$$\neg\, \texttt{f}\ (\texttt{y}\ /_{\texttt{b}}\ 0^{32})$$

For all environments E, it holds that

$$[\![\texttt{y}\ /_{\texttt{b}}\ 0^{32}]\!]_{\mathrm{SMT}}\, E\ =\ u$$

Therefore the expressions are equivalent for any assignment to y in SMT and the framework will report t as a correct implementation of s.
In IL/F the first program always terminates with u and the second program crashes. Therefore, t cannot be a correct implementation f s. We want to construct the formulas such that the solver can never prove for such a pair of programs that t is a correct implementation of s. On an intuitive level this is necessary as the optimization that modified the source program changed the programs termination behavior. For the target program we add the constraint that the divisor, in this case 0^{32}, cannot be assigned to 0^{32} as an implication:

$$\neg\, 0^{32} = 0^{32}\ \rightarrow\ \neg\, \texttt{f}\ (\texttt{y}\ /_{\texttt{b}}\ 0^{32})$$

The solver cannot satisfy the first constraint. This makes the implication trivially satisfiable. Therefore if no guard in the source program was unsatisfiable, the whole formula is always satisfiable if the divisor may be 0^{32}. Thereby t is not a correct implementation of s as intended.

5.3 Relation to Undefined Behavior

The C-Standard [8] describes undefined behavior as the point where no restriction on the semantics of the program are imposed. A wide range of operations exist which can lead to undefined behavior according to the C-Standard. An example of such undefined behavior is a division by zero. By adding the guards as explained before, our framework agrees with this specification.

5.4 Formalization of Guard Generation

As in the examples, the translation function must add the guards in place before the corresponding statement. Guards are generated during the translation to an SMT formula. First we define

how to generate a guard for an expression:

$$\begin{aligned}
\text{undef}\,(e_1\ /_b\ e_2) &= \neg \mathtt{e_2} = \mathtt{0^k} \ \wedge\ \text{undef}\,(e_1)\ \wedge\ \text{undef}\,(e_2) \\
\text{undef}\,(e_1\ \square\ e_2) &= \text{undef}\,(e_1)\ \wedge\ \text{undef}\,(e_2) \qquad\qquad \square \in \{\, +_b,\, -_b,\, *_b,\, =_b,\, \neq_b \,\} \\
\text{undef}\,(-_b\ e) &= \text{undef}\,(e) \\
\text{undef}\,(c) &= \mathtt{true} \\
\text{undef}\,(v) &= \mathtt{true}
\end{aligned}$$

The function *undef* works for any expression e that is in our bitvector language. It recursively computes guards for sub-expressions and combines them as conjunctions. We lift guard generation to lists of expressions, as our subset of IL/F includes function calls with argument lists.

$$\text{undef}\,(e_1, \ldots, e_n) \ =\ \text{undef}\,(e_1)\ \wedge\ \ldots\ \wedge\ \text{undef}\,(e_n)$$

The lifted version of *undef* takes as argument the argument list of a function call. The function produces the conjunction of the guards of each contained expression.
For large expressions, the guard may degenerate to a long conjunction of `true`. In the development we use the function *combine* instead of conjunctions to keeps the formulas as small as possible:

$$\begin{aligned}
\text{combine}\,(\mathtt{true}, \mathtt{true}) &= \mathtt{true} \\
\text{combine}\,(\mathtt{e_1}, \mathtt{true}) &= \mathtt{e_1} \\
\text{combine}\,(\mathtt{true}, \mathtt{e_2}) &= \mathtt{e_2} \\
\text{combine}\,(\mathtt{e_1}, \mathtt{e_2}) &= \mathtt{e_1}\ \wedge\ \mathtt{e_2}
\end{aligned}$$

We prove a lemma that *combine* is equivalent to $\wedge$:

Lemma 7
Let $\mathtt{e_1}$, $\mathtt{e_2}$ be two SMT formulas. Then:

$$F\ E\ \models\ \text{combine}\,(\mathtt{e_1}, \mathtt{e_2}) \Leftrightarrow F\ E\ \models\ \mathtt{e_1} \wedge \mathtt{e_2}$$

Proof.
The proof follows directly from definition of *combine* by a case distinction on whether $\mathtt{e_1} = \mathtt{true}$ and $\mathtt{e_2} = \mathtt{true}$.

□

By this lemma we justify that we we replace *combine* by $\wedge$ in this presentation.

5.5 Relation between Guards and Expression Evaluation

We formally relate satisfiability of the guard expressions to successful evaluations of expressions.

Lemma 8
Let V be a value environment, e a bitvector expression and v a bitvector constant. Then:

$$[\![e]\!]\, V\ =\ v \Longrightarrow\ \forall\ F,\ F\ \hat{V} \models \text{undef}\,(e)$$

and

$$[\![\overline{e}]\!]\, V\ =\ \overline{v} \Longrightarrow\ \forall\ F,\ F\ \hat{V} \models \text{undef}\,(\overline{e})$$

Proof.
The proof is by induction over the structure of the expression e. For the induction base assume first that e is a constant c. Then there is nothing to proof as $\text{undef}(c) = \texttt{true}$ and `true` is always satisfiable (M-TRUE). The base case where e is a variable is analogous.
In the induction step the case of a unary expression follows from the definition of *undef* and the inductive hypothesis.
Assume

$$e \equiv\ e_1 \,\square\, e_2$$

For each operator from our subset, except for $/_b$, we do a case distinction on whether e_1 evaluates and whether e_2 evaluates. If either of them does not evaluate, we have a contradiction. The case where both evaluate follows from the definition of *undef* and the inductive hypothesis.
We prove the case of a division $e \equiv e_1\ /_b\ e_2$ by contradiction. As before, if one of e_1 or e_2 crashes, we have a contradiction and close the proof. Assume both e_1 and e_2 evaluate. With the inductive hypothesis it holds that $\text{undef}(e_1)$ and $\text{undef}(e_2)$ are satisfiable. By definition

$$\text{undef}(e_1\ /_b\ e_2) = \neg \mathtt{e_2} = \mathtt{0^k} \wedge \text{undef}(e_1) \wedge \text{undef}(e_2)$$

We do a case distinction on

$$[\![e_2]\!]\, V\ =\ 0^{32}$$

If the value of e_2 is not 0^{32}, we close the prove with the inductive hypothesis and the lemma that partial evaluation implies total evaluation (Lemma 6). In the case where $[\![e_2]\!]\, V\ =\ 0^{32}$, we conclude by the definition of $/_b$:

$$[\![e]\!]\, V\ =\ \bot$$

This is a contradiction to the assumption that the expression evaluates to a value v.
The generalization of Lemma 8 to expression lists can be proven by a straight-forward induction over $\overline{e}$.

□

Next we relate satisfiability of $\text{undef}(e)$ for an environment $\hat{V}$ to evaluation of e.
Let the value environment V be defined on all variables that occur in e. If we know that $\text{undef}(e)$ can be modeled by any predicate environment F and the environment $\hat{V}$, e must evaluate under the value environment V.

Lemma 9
Let F be a predicate environment, V an IL/F value environment and e be a bitvector expression. If the free variables of e are bound in V then:

$$F\ \hat{V} \models \text{undef}(e) \Longrightarrow\ \exists\ \text{v}, [\![e]\!]\, V\ =\ v$$

and

$$F\ \hat{V} \models \text{undef}(\overline{e}) \Longrightarrow\ \exists\ \overline{v},\ [\![\overline{e}]\!]\, V\ =\ \overline{v}$$

Proof.
The proof is by induction over the expression e. For the base case, the case of a constant c trivially holds. The case of a variable follows from the assumption that the variables of e are defined in V.
For $e \equiv -_b\, e_1$ it holds that $\text{undef}(e) = \text{undef}(e_1)$. Then we can close the proof with the inductive

hypothesis and the definition of $-_b$.
For binary operators, except for $/_b$, the proof closes by the same argument. Let

$$e \equiv e_1 \ /_b \ e_2$$

Therefore it holds that

$$\text{undef}\,(e) = \neg\, \mathtt{e_2} = \mathtt{0^{32}} \wedge \text{undef}\,(e_1) \wedge \text{undef}\,(e_2)$$

As

$$F\ \hat{V} \models \text{undef}\,(e_2)$$

we conclude from the inductive hypothesis that

$$\exists v_2, [\![e_2]\!]\, V \ = \ v_2$$

By the lemma that partial evaluation implies total evaluation (Lemma 6) we conclude

$$[\![e_2]\!]_{\text{SMT}}\, \hat{V} \ = \ v_2$$

As we know that $F\ \hat{V} \models \text{undef}\,(e)$ we conclude

$$F\ \hat{V} \ \models \neg\, \mathtt{e_2} = \mathtt{0^{32}}$$

By M-Neg and M-Constr we conclude

$$v_2 \neq 0^{32}$$

Therefore, the division $e_1 \ /_b \ e_2$ is defined and

$$\exists v, [\![e]\!]\, V \ = \ v$$

The generalization of Lemma 9 to expression lists $\overline{e}$ is proven by a straight-forward induction over $\overline{e}$.

□

For the next lemma, assume that the IL/F statement s contains no function definitions and no external function calls. Assume s leaf-evaluates until the value statement e. If V' is the environment when reaching the return statement, then for all predicate environments F, $\text{undef}\,(e)$ is satisfiable with the environment V'.

Lemma 10
Let $(L \mid V \mid s)$ and $(L \mid V' \mid e)$ be IL/F states, where e ranges over expressions. Then:

$$(L \mid V \mid s) \ \Downarrow^t \ (L' \mid V' \mid e) \Longrightarrow \ \forall\ F,\ F\ \hat{V'} \models \text{undef}\,(e)$$

Proof.
By inversion on $\Downarrow^t$ it holds that

$$\exists v, [\![e]\!]\, V' \ = \ v$$

The fact that expression evaluation implies guard satisfiability (Lemma 8) finishes the proof.

□

A similar lemma can be defined for function calls. The returned expression is an expression list and the guard therefore becomes a conjunction of guards for the expressions of the list.

Lemma 11
Let $(L \mid V \mid s)$ and $(L \mid V' \mid f\,(\overline{e}))$ be IL/F states, where $\overline{e}$ is a list of bitvector expressions. Then:

$$(L \mid V \mid s) \;\Downarrow^t\; (L' \mid V' \mid f\,(\overline{e})) \Longrightarrow\; \forall\, F,\; F\;\hat{V'} \models \mathrm{undef}\,(\overline{e})$$

Proof.
The proof is analogous to the proof of Lemma 10.

□

We end this section with a lemma opposite to the lemma that expression evaluation implies satisfiability (Lemma 8). Let the value environment V be defined on the free variables of e. Assume that e does not evaluate under V. We show that the guard expression $\mathrm{undef}\,(e)$ cannot be satisfied by the environment $\hat{V}$ and any predicate environment F.

Lemma 12
Let F be a predicate environment, V a value environments and e a bitvector expressions.
If V is defined on the free variables of e then

$$[\![e]\!]\,V \;=\; \bot \Longrightarrow\; F\;\hat{V} \not\models \mathrm{undef}\,(e)$$

and

$$[\![\overline{e}]\!]\,V \;=\; \bot \Longrightarrow\; F\;\hat{V} \not\models \mathrm{undef}\,(\overline{e})$$

Proof.
The proof is by induction over the expression e.
For the base cases of constants or variables, we can establish a contradiction to the assumption that the expression does not evaluate. For the induction step we begin with negation:

$$e \;\equiv\; -_b\, e_1$$

We distinguish whether e_1 evaluates or not. Assume first that

$$\exists\; v,\; [\![e_1]\!]\,V \;=\; v$$

As evaluation of the unary operator $-_b$ cannot crash, we have established a contradiction to the assumption that e does not evaluate. So assume that e_1 does not evaluate. By the inductive hypothesis we know that $\mathrm{undef}\,(e_1)$ is not satisfiable. As we know that $\mathrm{undef}\,(-_b\; e_1) = \mathrm{undef}\,(e_1)$ we can then conclude that

$$F\;\hat{V} \not\models \mathrm{undef}\,(-_b\; e_1)$$

Let e be a binary expression:

$$e \;\equiv\; e_1 \Box\; e_2$$

Assume $\Box$ is not $/_b$. If e_1 and e_2 both evaluate, we have again a contradiction to the assumption that e does not evaluate. If either of e_1 and e_2 crashes, we conclude by the inductive hypothesis that there exists a guard that is not satisfiable. By definition of *undef* we can then close the proof.
Assume

$$e \equiv e_1\; /_b\; e_2$$

Assume that e_1 and e_2 evaluate. Otherwise we close the proof with the inductive hypothesis as before. With the assumption

$$[\![e]\!]\,V \;=\; \bot$$

we conclude by the definition of $/_b$ that

$$[\![e_2]\!]\, V \;=\; 0^{32}$$

By the lemma that partial evaluation implies total evaluation (Lemma 6) it holds

$$[\![e_2]\!]_{\mathrm{SMT}}\, \hat{V} \;=\; 0^{32}$$

The guard for e is

$$\mathrm{undef}\,(e_2) = \neg\, \mathtt{e_2} = \mathtt{0}^{32} \;\wedge\; \mathrm{undef}\,(e_1) \wedge \mathrm{undef}\,(e_2)$$

As we know that the value of e_2 is 0^{32} the constraint $\neg\, \mathtt{e_2} = \mathtt{0}^{32}$ cannot be satisfiable. Therefore

$$F\ \hat{V} \not\models \mathrm{undef}\,(e)$$

holds.
Lemma 12 is lifted to lists by a straight-forward induction over $\overline{e}$.

□

6 Formula Generation

In this section we define the translation of an IL/F program into an SMT formula. We assume that every pair of IL/F programs is renamed apart from each other.

6.1 Overview of the Conversion

The difference between the conversions for source and target programs lies in the generation of the last literal of the formula and how the guards are added.

arithmetic expressions Whenever the translation function encounters an arithmetic expression e, it generates the guard $\mathrm{undef}(e)$. This guard is prepended to the formula of the IL/F statement that is translated. For the source translation, the guard is added as conjunction and for the target translation, the guard is added as an implication.

conditionals For conditionals we exploit that SMT solvers support an `ite` clause whose semantics resemble those of a conditional in IL/F.

variable binding In IL/F a variable binding binds the left hand side of the assignment to the value of the right hand side. In SMT we need a similar structure as every use of the left-hand side of the variable binding should have the same value as the right-hand side in the formula. This can be achieved with constraints.

function calls Evaluation stops at function calls in our subset. Hence our formula generation must terminate at function calls, too. As SMT formulas support uninterpreted function symbols we simply introduce a predicate for every function symbol that the program contains. The SMT predicate has the same arity as the IL/F function. In the target translation, the literal is negated. As we do not impose any constraints on the function environment, it is possible that two programs may contain calls to the same function with different arity. Analysing the structure of function calls belongs to checking loop invariants. The framework does not handle loop invariants. Therefore the framework rejects any optimization that changes behavior of function calls. In subsection 9.3 we give an outline of how the formulas may be translated in a format compliant to the QF_UFBV logic.

value statements As we use uninterpreted function symbols, we add a special value predicate such that the case of a value statement reduces to the case of a function call. We use `ret` to denote this predicate.

6.2 Formalization of the Conversion

The translation of IL/F programs to SMT formulas does structural recursion on the IL/F statement that is converted. It traverses the program and concatenates the sub-formulas for the different statements. We define the translation to SMT for IL/F statements:

$$
\begin{aligned}
[\mathit{let}\ x\ =\ e\ \mathit{in}\ s]^{+} &= \mathrm{undef}(e)\ \wedge\ \mathtt{x} = \mathtt{e}\ \wedge\ [s]^{+} \\
[\mathit{let}\ x\ =\ e\ \mathit{in}\ s]^{-} &= \mathrm{undef}(e)\ \rightarrow\ \mathtt{x} = \mathtt{e}\ \wedge\ [s]^{-} \\
[\mathit{if}\ e\ \mathit{then}\ s_1\ \mathit{else}\ s_2]^{+} &= \mathrm{undef}(e)\ \wedge\ \mathtt{ite}\ (\mathtt{e})\ [s_1]^{+}\ [s_2]^{+} \\
[\mathit{if}\ e\ \mathit{then}\ s_1\ \mathit{else}\ s_2]^{-} &= \mathrm{undef}(e)\ \rightarrow\ \mathtt{ite}\ (\mathtt{e})\ [s_1]^{-}\ [s_2]^{-} \\
[f\ (\overline{e})]^{+} &= \mathrm{undef}(e)\ \wedge\ \mathtt{f}\ (\overline{\mathtt{e}}) \\
[f\ (\overline{e})]^{-} &= \mathrm{undef}(e)\ \rightarrow\ \neg\ \mathtt{f}\ (\overline{\mathtt{e}}) \\
[e]^{+} &= \mathrm{undef}(e)\ \wedge\ \mathtt{ret}\ (\mathtt{e}) \\
[e]^{-} &= \mathrm{undef}(e)\ \rightarrow\ \neg\ \mathtt{ret}\ (\mathtt{e})
\end{aligned}
$$

6.3 Soundness of Translation Validation

The rest of the thesis will be concerned with the proof of the following soundness property:

Theorem 1
Let s and t be IL/F programs that are renamed apart from each other. Assume that s and t contain no function definitions or external function calls. Let V be defined on the free variables of s and t. Then:

$$\not\models\ [s]^{+}\ \wedge\ [t]^{-}\ \Longrightarrow\ (L \mid V \mid s) \lesssim (L \mid V \mid t)$$

7 Proof of the Soundness Theorem

In this section we prove the soundness theorem for the translation validation algorithm. We first give an overview of the structure:
For all programs in our subset of IL/F, we have proven that it is decidable whether they leaf-evaluate or leaf-crash (Lemma 3). Therefore, we do a case distinction according to this lemma for s.

- Let s leaf-evaluate.
  We do the same case distinction on whether the target t leaf-evaluates or leaf-crashes (Lemma 3).
  - Let t leaf-evaluate. We show in subsection 7.2 with Lemma 14, that in this case, both programs terminate with the same value (or issue the same function call).
  - Let t leaf-crash. We show in subsection 7.5 with Lemma 17 that this is a contradiction to the assumption that the formula $[s]^+ \wedge [t]^-$ is satisfiable and can derive a proof of $\bot$.
- Let s leaf-crash. According to the definition of $\lesssim$, if s crashes, the compiler can produce any program when optimizing s. Therefore any optimization result is a correct implementation of s.

7.1 Rewriting Unsatisfiable Formulas

To prove the case where both s and t terminate an additional lemma is needed to ease the proof. Currently we do not know anything about the structure of s or the structure of t. The equivalence proof hence involves induction on the structure of s and t.
Assume a program s leaf-evaluates until *s'*.

$$(L \mid V \mid s) \Downarrow^t (L \mid V' \mid s') \tag{4}$$

Let $p \in \{+, -\}$ be a parameter denoting any possible SMT translation. Whenever we know that for any C

$$\not\models [s]^p \wedge C$$

holds for any p, replacing s with s' may make the formula satisfiable:

$$\exists\, F\ E, F\ E \models [s']^p \wedge C$$

Take for example

```
let x = 5 in
f x
```

The formula for the source translation of this program is

$$\texttt{x} = \texttt{5} \ \wedge\ \texttt{f}\ (\texttt{x})$$

The guard expression `true` was omitted for readability. Let C be $\neg\, \texttt{x} = \texttt{5}$. The combined formula is

$$\texttt{x} = \texttt{5} \ \wedge\ \texttt{f}\ (\texttt{x}) \ \wedge\ \neg\, \texttt{x} = \texttt{5}$$

This formula is unsatisfiable.
It holds that

$$(L \mid V \mid \mathit{let}\ x\ =\ 5\ \mathit{in}\ f\,(x)) \Downarrow^t (L \mid V[x \leftarrow 5] \mid f\,(x))$$

Intuitively, the evaluation step moves the variable binding introduced by the let-statement from the program text to the value environment. Replacing the formula for Listing 7.1 with the formula for $f\,(x)$ removes the information that x is bound to 5. This yields a satisfiable formula:

$$\texttt{f (x)} \ \wedge \ \neg\, \texttt{x} = \texttt{5}$$

To preserve unsatisfiability it is necessary to add a constraint that resembles the fact that x needs to be bound to 5. The constraint $\texttt{x} = \texttt{5}$ is exactly this constraint.
Now look at

```
if x = 5
  then f x
  else g x
```

The formula for the source translation of this program is

$$\texttt{ite (x} = \texttt{5) (f (x)) (g (x))}$$

The guard expression `true` was omitted for readability. With $\mathrm{C} \equiv \texttt{ite (x} = \texttt{5) } (\neg\, \texttt{f (x)}) \ (\neg\, \texttt{g (x)})$ the combined formula is unsatisfiable:

$$\texttt{ite (x} = \texttt{5) (f (x)) (g (x))} \ \wedge \ \texttt{ite (x} = \texttt{5) } (\neg\, \texttt{f (x)}) \ (\neg\, \texttt{g (x)})$$

Assume that

$$(L \mid V \mid \mathit{if}\ x = 5\ \mathit{then}\ f\,(x)\ \mathit{else}\ g\,(x)) \Downarrow^{t} (L \mid V \mid f\,(x))$$

There is no new value binding introduced in V. But the reduction to the statement $f\,(x)$ implicitly contains the information that for the environment V the condition $x = 5$ is true. Replacing the formula for Listing 7.1 with the formula for $f\,(x)$ yields again a satisfiable formula:

$$\texttt{f (x)} \ \wedge \ \texttt{ite (x} = \texttt{5) } (\neg\, \texttt{f (x)}) \ (\neg\, \texttt{g (x)})$$

To keep the formula unsatisfiable we need to add an SMT constraint. This constrain must resemble the knowledge that the condition must evaluate to `true`:

$$\texttt{ite (x} = \texttt{5) (true) (false)}$$

This `ite`-constraint encodes the same information about the value environment as the IL/F conditional and does not introduce any additional constraints on the used variables. Adding this `ite`-constraint makes the formula unsatisfiable again:

$$\texttt{ite (x} = \texttt{5) (true) (false)} \ \wedge \ \texttt{f (x)} \ \wedge \ \texttt{ite (x} = \texttt{5) } (\neg\, \texttt{f (x)}) \ (\neg\, \texttt{g (x)})$$

Assume a program s, that is in our subset of IL/F leaf-evaluates until *s'*. Remember that $p \in \{+, -\}$. As in the examples, we can find a formula `Q` such that the conjunction of `Q` and the formula $[s']^p$ is still unsatisfiable. This is exactly the case for all `Q` that represent the control flow and data flow of s before reaching *s'*. A mathematical definition of this lemma is:

Lemma 13
Let $(L \mid E \mid s)$,$(L \mid E' \mid s')$ be IL/F states and $p \in \{+, -\}$ an SMT translation,
there is a Q with $\mathrm{Vars}(Q) \subseteq \mathrm{Vars}(s)$, such that

$$\begin{aligned} & \not\models \ [s]^p \wedge C \\ \Longrightarrow \ & (L \mid V \mid s) \Downarrow^{t} (L \mid V' \mid s') \\ \Longrightarrow \ & \forall\, F,\, F\ \hat{V}' \models Q \\ \Longrightarrow \ & \not\models \ Q \wedge [s']^p \wedge C \end{aligned}$$

The proof of Lemma 13 is by induction over the statement s. In the base cases of value statements or function calls it suffices to choose `true` for the existential as s is the same as s'.
For the induction step the cases of function definitions and external function calls do not occur as they are not in the subset of the framework. The cases of variable bindings and conditionals are generalized versions of the examples from this chapter.

7.2 s Leaf-evaluates and t Leaf-evaluates

We have shown with Lemma 2 that whenever a state $(L \mid V \mid s) \Downarrow^t (L \mid V' \mid s')$ then s' is either a function call or a value statement. As explained in Section 6 a value statement is handled as a special function call. Therefore we assume for the next lemma that whenever $(L \mid V \mid s) \Downarrow^t (L \mid V' \mid s')$ holds, then s' is a function call. The lemma for value statements follows by an analogous argument.

Lemma 14
Let s and t be two IL/F programs. Assume the value environment V and the label environment L. Then:

$$\begin{aligned} & \not\models [s]^+ \wedge [t]^- \\ \Rightarrow\ & (L \mid V \mid s) \Downarrow^t (L \mid V_s \mid f\,(\overline{e_s})) \\ \Rightarrow\ & (L \mid V \mid t) \Downarrow^t (L \mid V_t \mid g\,(\overline{e_t})) \\ \Rightarrow\ & f = g \wedge [\![\overline{e_s}]\!]\,V_s = [\![\overline{e_t}]\!]\,V_t \end{aligned}$$

Proof.
Applying Lemma 13 twice to the assumption that the formulas are unsatisfiable we get

$$\not\models \mathtt{Q_s} \wedge \text{undef}\,(\overline{e_s}) \wedge \mathtt{f}\ (\overline{\mathtt{e_s}}) \wedge \mathtt{Q_t} \wedge \text{undef}\,(\overline{e_t}) \wedge \neg\,\mathtt{g}\ (\overline{\mathtt{e_t}})$$

Under which conditions can t' be a correct implementation of s'?
From Lemma 13 we know that

$$\forall\ F,\ F\ \hat{V_s} \models Q_s$$

and

$$\forall\ F,\ F\ \hat{V_t} \models Q_t$$

Therefore we first define the combined value environment V_c:

$$V_c\,x = \begin{cases} V_s\,x & \text{if x} \in \text{Vars}(s) \\ V_t\,x & \text{otherwise} \end{cases}$$

The programs s and t are renamed apart from each other. Therefore V_c gives the correct values for the defined variables. For the free variables V_c will always return the value from V_s. All free variables can never be rebound in any program due to the α-conversion step. The executions for s and t start with the same environment V. Combining these two facts yields that whenever for a free variable x

$$V_s\,x = v$$

then it must hold that

$$V_t\,x = v$$

Variables that are not bound in V are mapped to the bitstring 0^{32} as we have to lift V_c to a total environment.

By the lemma that leaf-evaluation implies guard satisfiability (Lemma 11), the guards $\text{undef}\,(\overline{e_s})$ and $\text{undef}\,(\overline{e_t})$ are satisfiable by $\hat{V}_s$, respectively $\hat{V}_t$.
Due to Lemma 13 the variables in $\mathtt{Q_s}$ can only come from the variables of s. For the same reason the variables of $\mathtt{Q_t}$ can only come from the variables of t. As s and t are renamed apart from each other all constraints in $\mathtt{Q_s}$ and $\mathtt{Q_t}$ argue about different variables and we know that

$$\forall\, F,\, F\ \hat{V}_c\ \models\ \mathtt{Q_s}\ \wedge\ \mathtt{Q_t}$$

The intuition that equivalence can only hold if both *s'* and *t'* are a function call to the same function with the same arguments follows by contradiction to the assumption:

$$\not\models\ \mathtt{Q_s}\ \wedge\ \text{undef}\,(\overline{e_s})\ \wedge\ \mathtt{f}\ (\overline{\mathtt{e_s}})\ \wedge\ \mathtt{Q_t}\ \wedge\ \text{undef}\,(\overline{e_t})\ \wedge\ \neg\,\mathtt{g}\ (\overline{\mathtt{e_t}})$$

How is this contradiction established?
Define the predicate environment that maps all predicates to `true` except for g. g is mapped to `false`, for all parameters:

$$\mathrm{F}\ :=\ (\lambda\ h.\ \lambda\ \bar{x}.\ \mathtt{true})\ [g\ \mapsto\ \lambda\ \bar{y}.\ \mathtt{false}]$$

Therefore it holds that

$$F\ \hat{V}_c\ \models\ \mathtt{Q_s}\ \wedge\ \text{undef}\,(\overline{e_s})\ \wedge\ \mathtt{f}\ (\overline{\mathtt{e_s}})\ \wedge\ \mathtt{Q_t}\ \wedge\ \text{undef}\,(\overline{e_t})\ \wedge\ \neg\,\mathtt{g}\ (\overline{\mathtt{e_t}})$$

This is a contradiction to the assumption that the conjunction of both formulas is unsatisfiable. Therefore we can conclude that both must be a call to the same function.
What remains to show is that the argument lists for the function call cannot differ. We show that this by contradiction. Assume *s'* is

$$f\,(\overline{e_s})$$

and *t'* is

$$f\,(\overline{e_t})$$

As both programs leaf-evaluate, the evaluation of $\overline{e_s}$ and $\overline{e_t}$ is defined. Therefore assume

$$[\![\overline{e_t}]\!]\,\hat{V}_c\ =\ \overline{v_t}$$

Define the predicate environment that maps all predicates to `true` for any argument list $\overline{a}$ except for f which maps $\overline{v_t}$ to `false` and all other arguments to `true`:

$$\mathrm{F}\ :=\ (\lambda\ h.\ \lambda\ \bar{x}.\ \mathtt{true})[f\ \mapsto\ \lambda\,\bar{x}.\text{if}\ \overline{x} = v_t\ \text{then}\ \mathtt{false}\ \text{else}\ \mathtt{true}]$$

Therefore it holds that:

$$F\ \hat{V}_c\ \models\ \mathtt{Q_s}\ \wedge\ \text{undef}\,(\overline{e_s})\ \wedge\ \mathtt{f}\ (\overline{\mathtt{e_s}})\ \wedge\ \mathtt{Q_t}\ \wedge\ \text{undef}\,(\overline{e_t})\ \wedge\ \neg\,\mathtt{f}\ (\overline{\mathtt{e_t}})$$

This is again a contradiction to the assumption that the conjunction of both formulas is unsatisfiable. From this we can conclude that the predicates must agree on the values for their parameters. Thereby the two programs do the IL/F function calls.
The case where s or t is a value statement closes by the same argument. Either both are a a value statement with the same result value or we establish a contradiction to the assumption. Therefore the programs s and t compute the same results.

□

To prove the last remaining lemma, to close the proof of Theorem 1, we prove two lemmata that relate leaf-termination and leaf-crashing to formula satisfiability.

7.3 Leaf-evaluation and Satisfiability

Assume with the value environment V an IL/F program s terminates with a value environment V'. Furthermore let s be renamed apart. We prove that the source translation of s is satisfiable for the value environment V' and the predicate environment that maps every function for every argument to `true`:

$$(\lambda\ \mathtt{f}.\ \lambda\ \overline{\mathtt{x}}.\ \mathtt{true})$$

Lemma 15
Let s and s' be IL/F programs that are renamed apart and let L be a function environment. Let V and V' be a value environment. Then:

$$(L \mid V \mid s) \Downarrow^t (L \mid V' \mid s') \Rightarrow (\lambda\ \mathtt{f}.\ \lambda\ \overline{\mathtt{x}}.\ \mathtt{true})\ \hat{V'} \models [s]^+$$

Proof.
The proof is by induction over

$$(L \mid V \mid s) \Downarrow^t (L \mid V' \mid s')$$

For the induction base, the case where T-Ret holds can be reduced to the case of T-Goto because we handle a value statements as a special function call.
Assume T-Goto holds. Therefore let s be a function call:

$$s \equiv f\,(e_1 \ldots e_n)$$

The translation to SMT of s is:

$$[f\,(e_1 \ldots e_n)]^+ \equiv \mathrm{undef}\,(e_1 \ldots e_n) \wedge \mathtt{f}\ (\mathtt{e_1} \ldots \mathtt{e_n})$$

By inversion on T-Goto we conclude that

$$\exists\ v_1 \ldots v_n,\ [\![e_1 \ldots e_n]\!]\,V = v_1 \ldots v_n$$

With the lemma that expression evaluation implies guard satisfiability (Lemma 8) we know that

$$(\lambda\ \mathtt{f}.\ \lambda\ \overline{\mathtt{x}}.\ \mathtt{true})\ \hat{V'} \models \mathrm{undef}\,(e_1 \ldots e_n)$$

holds.
For M-And to hold we show that

$$(\lambda\ \mathtt{f}.\ \lambda\ \overline{\mathtt{x}}.\ \mathtt{true})\ \hat{V'} \models \mathtt{f}\ (\mathtt{e_1} \ldots \mathtt{e_n})$$

By the lemma that partial evaluation implies total evaluation (Lemma 6) we conclude that

$$[\![\mathtt{e_1} \ldots \mathtt{e_n}]\!]_{\mathrm{SMT}}\,\hat{V} = \mathtt{v_1} \ldots \mathtt{v_n}$$

Therefore, by M-Pred

$$(\lambda\ \mathtt{f}.\ \lambda\ \overline{\mathtt{x}}.\ \mathtt{true})\ \mathtt{f}\ (\mathtt{e_1} \ldots \mathtt{e_n}) = \mathtt{true}$$

trivially holds by the definition of the predicate environment. Hence the induction base holds.
For the induction step we know that T-Step holds. By inversion on T-Step it holds that

$$(L \mid V \mid s) \xrightarrow{a} (L \mid V'' \mid s'')$$

With an inversion on this hypothesis we get first that s is a let-statement:

$$s \equiv \mathit{let}\ x = e\ \mathit{in}\ s'$$

Then

$$[\mathit{let}\ x = e\ \mathit{in}\ s']^+ \equiv \mathrm{undef}(e) \wedge \mathtt{x} = \mathtt{e} \wedge [s']^+$$

As s terminates, we conclude that e evaluates to a value v. With Lemma 8 we conclude that

$$(\lambda\ \mathtt{f}.\ \lambda\ \overline{\mathtt{x}}.\ \mathtt{true})\ \hat{V} \models \mathrm{undef}(e)$$

By the lemma that V and V' agree on the values for the free variables of s (Lemma 1), we replace V with V'. By M-And it remains to show that

$$(\lambda\ \mathtt{f}.\ \lambda\ \overline{\mathtt{x}}.\ \mathtt{true})\ \hat{V'} \models \mathtt{x} = \mathtt{e} \wedge [s']^+$$

By the inductive hypothesis

$$(\lambda\ \mathtt{f}.\ \lambda\ \overline{\mathtt{x}}.\ \mathtt{true})\ \hat{V'} \models [s']^+$$

holds. With Lemma 1 we conclude that we can exchange V with V' again and that the value of e is v in V'. By Lemma 6 we conclude that

$$[\![\hat{V'}]\!]_{\mathrm{SMT}}\, e = v$$

By the same argument for $V\,[x \mapsto e]$ and V' we conclude that the value of x is v. Hence the values of x and e agree in V'. By M-Constr we can conclude

$$(\lambda\ \mathtt{f}.\ \lambda\ \overline{\mathtt{x}}.\ \mathtt{true})\ \hat{V'} \models x = e$$

In the next case, assume $s \equiv \mathit{if}\ e\ \mathit{then}\ s_1\ \mathit{else}\ s_2$. The formula for s is then

$$[s]^+ \equiv \mathrm{undef}(e) \wedge \mathtt{ite}\ (\mathtt{e})\ ([\mathtt{s}_1]^+)\ ([\mathtt{s}_2]^+)$$

We want to prove that

$$(\lambda\ \mathtt{f}.\ \lambda\ \overline{\mathtt{x}}.\ \mathtt{true})\ \hat{V'} \models \mathrm{undef}(e) \wedge \mathtt{ite}\ (\mathtt{c})\ ([\mathtt{s}_1]^+)\ ([\mathtt{s}_2]^+)$$

Analogue to let-statements, we conclude that the guard $\mathrm{undef}(e)$ can be modeled. Therefore we have to prove that:

$$(\lambda\ \mathtt{f}.\ \lambda\ \overline{\mathtt{x}}.\ \mathtt{true})\ \hat{V'} \models \mathtt{ite}\ (\mathtt{c})\ ([\mathtt{s}_1]^+)\ ([\mathtt{s}_2]^+)$$

By Lemma 1 we conclude that we can exchange V with V' and with Lemma 6 we conclude

$$[\![c]\!]\, V = [\![c]\!]_{\mathrm{SMT}}\, \hat{V'}$$

Therefore we conclude that whenever the IL/F program enters one of the two branches the SMT formula will do the same. With a case distinction over the value of c we close this case with the inductive hypothesis. The cases of function definitions and external function calls do not occur as they are not handled in our framework.

□

7.4 Leaf-crashing and Satisfiability

We can prove a similar lemma on crashing program evaluations for target translations. Due to the design of the guard formulas, we can prove a more general form that holds for all predicate environments F. We first give an intuition why this lemma is possible. Assume

$$(L \mid V \mid s) \;\Downarrow^c\; (L \mid V' \mid s')$$

The statement s' crashes, hence it contains an expression e that does not evaluate under the value environment V'. We do target translation of s. Therefore, the framework produces guard formulas that are unsatisfiable if and only if the evaluation of the expression that is guarded is not defined in IL/F. Guard formulas are prepended such that whenever the guard is unsatisfiable, the whole subformula is satisfiable. As the evaluation crashes at s', every statement before s' did not crash, hence the formulas for these statements are satisfiable. Furthermore the guard for the expression e for which the evaluation fails must be unsatisfiable. This makes the implication that contains this last guard trivially satisfiable.
Assume a renamed apart IL program s. Let s, evaluated under the value environment V, crash with the value environment V' at the statement s'. Assume that V is defined on the free variables of s. We prove, that for every predicate environment, the translation of s as a target program is satisfiable under the value environment V'.

Lemma 16
Let s and s' be renamed apart IL/F programs and let V and V' be value environments. Let L be a function environment and F a predicate environment. Then:

$$(L \mid V \mid s) \;\Downarrow^c\; (L \mid V' \mid s') \Rightarrow \;\; F\ \hat{V} \models [s]^-$$

Proof.
The proof is by induction over

$$(L \mid V \mid s) \;\Downarrow^c\; (L \mid V' \mid s')$$

For the induction base assume C-Goto holds. We conclude that

$$s \;\equiv\; f\,(e_1 \ldots e_n)$$

We have to prove that

$$F\ \hat{V'} \;\models\; \mathrm{undef}\,(e_1 \ldots e_n) \rightarrow\ \neg f\,(\overline{x})$$

By C-Goto we conclude

$$[\![e_1 \ldots e_n]\!]\,V \;=\; \bot$$

We have proven that whenever an expression does not evaluate under an environment, then the guard is unsatisfiable for this environment (Lemma 12). Therefore, we know that the guard $\mathrm{undef}\,(e_1 \ldots e_n)$ is unsatisfiable. With this we know that the implication is trivially satisfiable.
C-Base follows by a similar argument. For each possible s in our subset, if we know

$$\forall \sigma, \neg\ (L \mid V \mid s) \rightarrow \sigma$$

then the evaluation of an expression e in s does not succeed. By Lemma 12, there is a guard $\mathrm{undef}\,(e)$ that is unsatisfiable. As we do target translation, the guard is added as an implication. Therefore the implication

$$\mathrm{undef}\,(e) \;\rightarrow\; [s]^-$$

is trivially satisfiable.
For the induction step, we know that C-Step holds. By inversion on C-Step it holds that

$$(L \mid V \mid s) \xrightarrow{a} (L \mid V'' \mid s'')$$

Assume

$$s \equiv \mathit{let}\ x = e\ \mathit{in}\ s'$$

We show that prepending the constraint $\mathtt{x} = \mathtt{e}$ preserves satisfiability. Satisfiability of s' follows by the inductive hypothesis. As

$$(L \mid V \mid \mathit{let}\ x = e\ \mathit{in}\ s') \xrightarrow{\tau} (L \mid V\,[x \mapsto v] \mid s')$$

we conclude that

$$\exists\, v,\ [\![e]\!]\, V = v$$

By the lemma that renamed apart leaf-crashing programs cannot overwrite introduced bindings (Lemma 1) we conclude that we can exchange V and V'. Hence we conclude that

$$[\![e]\!]\, V = [\![e]\!]\, V'$$

By the same argument, we conclude that

$$V\,[x \mapsto v]\, x = V'\, x$$

Therefore we conclude

$$V'\, x = [\![e]\!]\, V$$

As all expressions we argue about evaluate, we can trivially lift the assumptions to total environments with Lemma 6. Therefore the constraint $\mathtt{x} = \mathtt{e}$ is satisfiable. Satisfiability of $[s']^{-}$ follows from the inductive hypothesis.
Assume

$$s \equiv \mathit{if}\ e\ \mathit{then}\ s_1\ \mathit{else}\ s_2$$

We repeat the argument from the case of a let-statement. As we know that there exists a state σ such that

$$\mathit{if}\ e\ \mathit{then}\ s_1\ \mathit{else}\ s_2 \xrightarrow{\tau} \sigma$$

we conclude that e evaluates to a value v. Again with Lemma 1 we conclude

$$[\![e]\!]\, V = [\![e]\!]\, V'$$

By Lemma 6 we lift the value again to the total environment $\hat{V}'$. With a case distinction over

$$\mathbb{B}(e) = i$$

we can distinguish whether the program enters s_1 or s_2 and close the proof with the inductive hypothesis. The cases of function definitions and external function calls do not occur as they are not handled in our framework.

□

7.5 s Leaf-evaluates and t Crashes

Lemma 17
Let s and t be two IL/F programs. Assume the value environment V and the label environment L. Then:

$$\begin{aligned} & \not\models [s]^+ \wedge [t]^- \\ \Rightarrow\ & (L \mid V \mid s) \Downarrow^t (L \mid V_s \mid s') \\ \Rightarrow\ & (L \mid V \mid t) \Downarrow^c (L \mid Vt \mid t') \\ \Rightarrow\ & \bot \end{aligned}$$

Proof.
From Lemma 15 it follows that the formula $[s]^+$ is satisfiable for the environment V_s and the predicate environment that maps every predicate to `true`. From Lemma 16 it follows that the formula $[t]^-$ is satisfiable for the environment V_t and the predicate environment that maps every predicate to `true`. To get a contradiction we use again the combined environment V_c (subsection 7.2).
With V_c and the predicate environment that maps every predicate to `true` it holds that

$$(\lambda f.\, \lambda \bar{x}.\, \texttt{true}.)\ \hat{V_C} \ \models\ [s]^+ \wedge\ [t]^-$$

Hence we have established a contradiction.

□

8 Related Work

IL In his masters thesis, Schneider [20] describes the development of the LVC framework on which this thesis is based. In the thesis he has developed and proven correct a compiler for the IL intermediate language used to express programs in this thesis. In a recent tech report [22], the LVC framework has been extended by an implenentation relation for IL programs that is sound for diverging programs. In another tech report [21], the LVC framework has been extended by an approach to SSA[5] form construction based on a semantic characterization of SSA.

SMT Ranise and Tinelli [19] explain the foundations of SMT solvers. They give an introduction to the different approaches on how an SMT solver can be implemented with respect to the fact that it needs to support multiple theories.
Barrett et al. [2] give an overview of common SMT theories. In their text they explain ways of encoding SMT satisfiability in SAT as well as ways of deciding SMT satisfiability using theorem provers.
Barrett, Stump, and Tinelli [3] give a standardized way of communicating with SMT solvers, called SMT-Lib. Armand et al. [1] also integrated SAT/SMT solvers into Coq. Instead of using them in a specific use case like translation validation, they enabled validation of SMT solver responses inside of Coq. Therefore they enabled encoding of SMT solver proofs in Coq.
Li et al. [11] present their optimization algorithm SYMBA. SYMBA was implemented for the theory of Linear Real Arithmetic[3]. Instead of producing satisfying assignments, SYMBA takes as an input a formula and a variable that occurs in this formula. The tool tries to maximize this value in the satisfying assignments of the formula.
Gulwani et al. [7] use SMT solvers to synthesize loop-free programs given a propositional specification. Their framework is given a library of functions that may occur in the synthesized program. They produce an SMT formula from which they can reconstruct with an SMT solver whether there exists a combination of the library functions such that they compute the same result as the specified program.

Translation Validation Necula [15] developed a translation validation framework for optimizing C compilers. The framework handles most of the intra-procedural optimizations done in the GNU C compiler. Instead of using SMT solvers, his framework produces simulation relations relating program points in the source and the target program. They collect constraints that need to be satisfied for a pair of program points (s,t) where s is in the source and t in the target program. Equivalence in their framework is equivalence of observable behaviours.
Pnueli, Siegel, and Singerman [18] describes translation validation for a translation from the synchronous SIGNAL language to asynchronous C. Their intermediate representation are so called synchronous transition systems [17]. Refinement mappings are used as their simulation technique for observable behavior of source and target programs. They also implemented a proof checker which checks the equivalence proofs constructed by their intermediate representation analyzer.
Namjoshi and Zuck [14] describe an approach similar to translation validation. Each optimization in a compiler provides a witness in form of a stuttering simulation to correlate the source and the target program. Their focus was on additionally enabling use of the witnesses as invariants for the analyzed programs which can be propagated forward to optimizations as additional information. The witness generation needs to be implemented by the optimization developer for each optimization.
Stepp, Tate, and Lerner [23] present a Translation Validator for LLVM. Their intermediate representation are Program Expression Graphs [24]. They extend their tool Peggy [24] which is able to do so called equational saturation [24]. In their results it is mentioned that future work

may involve usage of SMT solvers to enhance their decision procedure. The tool Peggy was not proven correct. Instead they presented experimental results.
Mansky and Gunter [12] formalized Translation Validation in Isabelle/HOL. In the paper they modified the TRANS language [9] and verified it in Isabelle/HOL to be able to prove correctness of SSA construction for programs. Their used intermediate representation are control flow graphs. According to the paper TRANS describes conditional modifications of control flow graphs. The implementation of the modification algorithm and the verification were done with Isabelle/HOL. Our approach used Coq only to verify that the SMT based approach produces sound answers.

Undefined Behavior Brummayer and Biere [4] used SMT solvers to check whether a C expression contains undefined behavior or not. Their SAT based approach checks whether a C expression contains undefined behavior that may happen at runtime. Their framework works on a single C expression. We focus instead on whole programs and do not need to know whether the undefined behavior occurs. We only want to be able to produce the correct output if the undefined behavior may occur and are not interested in knowing where it occurred. They translate C expressions into AIG graphs[10] which are then bit blasted into a SAT problem. The AIG graphs have an additional bit to track whether their value is undefined.

9 Future Work

9.1 Effect Monads in SMT

Currently the development is restricted such that only conditionals, variable bindings and value statements as well as function calls are allowed. We disallowed external function calls as they require a more complex translation to SMT formulas. We give a short overview on how we hope to be able to integrate them into the development. Assume the program

$$let\ y\ =\ \alpha\,(\overline{x})\ in\ s'$$

Translating this program to SMT with our current approach leads a formula similar to

$$\mathtt{y} = \alpha\ (\overline{\mathtt{x}})\ \wedge [s']^p$$

The problem here lies in executions of α. As external function calls are also undefined in IL/F their effect on the variable y is undecidable. Furthermore, the behavior of an external function β could be influenced by the fact whether another external function γ has been executed before. One example of such behavior is *malloc* and $free$ in C.
We use *malloc* and *free* to illustrate necessary steps to model external function calls. Intuitively what the C standard states about *malloc* and *free* is that one should only *free* memory that has been allocated by *malloc* before. Furthermore, on no memory *free* may be called twice. So we must be able to encode somehow in the SMT formula the fact that an external function call must be executed before an other.
In SMT it is possible to define constructors for so called sorts. We hope to be able to exploit this to define effect monads that keep track of the execution order by adding them as parameters to an external function call and a result value.
Take again the example from before. Adding the effect monads m_1 and m_2 yields the formula

$$(\mathtt{m_2}, \mathtt{x}) = \alpha\ (\mathtt{m_1}, \overline{\mathtt{x}})\ \wedge\ [s']^p$$

Assume s' would contain another external function call β. The effect monad m_2 is added as a parameter to the function β. The return value of β is a pair that consists of the next effect monad m_3 and the value for the introduced variable.
A technical problem of this approach is that the IL/F semantics do not contain pairs as possible result values. Therefore the presented expression language would have to be extended by pairs for SMT expression. Thereby we would get two different expression languages where it would be necessary to prove that the IL/F expression language can be embedded in the new SMT language.
We hope that this does not involve any changes for the decision procedure.
Assume the solver is able to find a satisfying assignment for the extended formulas. We expect this to mean that there is a possible result for an external function call such that it returns a value that will make the program different from the second one. Assume the solver proves unsatisfiability of the formula. We hope to conclude that for every possible result of the external function call, the target program is a correct implementation of the source.

9.2 Correctness for Interprocedural Optimization runs

The restricted programs in this thesis can be seen as function bodies. We presented an approach to prove that one function body is a correct implementation of the other when no external function calls occur. We hope to be able to iteratively apply this approach for each defined

function body in the source and the target program. Thereby we would then establish that each function block in the target program is a correct implementation of the corresponding source block. Thereof we hope to conclude that the target program is a correct implementation of the source program.

9.3 Compliance with QF_UFBV

As explained in Section 4 our formalized logic is not directly compliant with a specified logic of an SMT solver. When translating to SMT the arity of an introduced uninterpreted function symbol could be memorized. With this it should be possible to account for the fact different arity of a function must be encoded as different function symbols. As each function name is only a natural number in IL/F two uses of a function symbol with different arity can then be made different by incrementing one of the two labels. With this modification, the logic does not need to express arbitrary argument lists anymore. The lengths of argument lists can then be statically determined. Hence the parameter lists of undefined length would not be necessary anymore. We hope that this makes the formulas compliant to the format of the QF_UFBV logic.

9.4 Combination with other approaches

In Section 8 we explained how Namjoshi and Zuck [14] have taken an approach similar to Translation Validation. We hope that in combination with our approach it is possible to prove correct more complex optimizations. For correctness of complex optimizations, we hope to be able to use the stuttering simulations used in their approach to correlate the optimized blocks. For blocks from our restricted subset, the fact that a target block is a correct implementation of a source block, can then be proven with the approach presented in this thesis.

10 References

[1] Michaël Armand et al. “A modular integration of SAT/SMT solvers to Coq through proof witnesses”. In: *Certified Programs and Proofs* (2011), pp. 135–150.

[2] Clark W Barrett et al. “Satisfiability Modulo Theories.” In: *Handbook of satisfiability* 185 (2009), pp. 825–885.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. “The SMT-LIB Standard: Version 2.0”. In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*. Ed. by A. Gupta and D. Kroening. 2010.

[4] Robert Brummayer and Armin Biere. “C32sat: Checking c expressions”. In: *Computer Aided Verification*. Springer Berlin Heidelberg. 2007, pp. 294–297.

[5] Ron Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pp. 451–490.

[6] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[7] Sumit Gulwani et al. “Synthesis of loop-free programs”. In: *ACM SIGPLAN Notices*. Vol. 46. 6. ACM. 2011, pp. 62–73.

[8] ISO/IEC. *ISO 9899:2011 - Programming languages - C*. Geneva, Switzerland, 2011.

[9] Sara Kalvala, Richard Warburton, and David Lacey. “Program transformations using temporal logic side conditions”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31.4 (2009), p. 14.

[10] Andreas Kuehlmann et al. “Robust Boolean reasoning for equivalence checking and functional property verification”. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 21.12 (2002), pp. 1377–1394.

[11] Yi Li et al. “Symbolic optimization with SMT solvers”. In: *ACM SIGPLAN Notices* 49.1 (2014), pp. 607–618.

[12] William Mansky and Elsa Gunter. “A framework for formal verification of compiler optimizations”. In: *Interactive Theorem Proving* (2010), pp. 371–386.

[13] Matthew W Moskewicz et al. “Chaff: Engineering an efficient SAT solver”. In: *Proceedings of the 38th annual Design Automation Conference*. ACM. 2001, pp. 530–535.

[14] Kedar S Namjoshi and Lenore D Zuck. “Witnessing program transformations”. In: *Static Analysis* (2013), pp. 304–323.

[15] George C Necula. “Translation validation for an optimizing compiler”. In: *ACM sigplan notices* 35.5 (2000), pp. 83–94.

[16] Behrooz Parhami. *Computer arithmetic*. Oxford University Press, 2000.

[17] Amir Pnueli, Natarajan Shankar, and Eli Singerman. “Fair synchronous transition systems and their liveness proofs”. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer. 1998, pp. 198–209.

[18] Amir Pnueli, Michael Siegel, and Eli Singerman. “Translation validation”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1998, pp. 151–166.

[19] Silvio Ranise and Cesare Tinelli. "Satisfiability modulo theories". In: *Trends and Controversies-IEEE Intelligent Systems Magazine* 21.6 (2006), pp. 71–81.

[20] Sigurd Schneider. "Semantics of an Intermediate Language for Program Transformations". Master's Thesis. Universität des Saarlandes, 2013.

[21] Sigurd Schneider, Gert Smolka, and Sebastian Hack. *A First-Order Functional Intermediate Language for Verified Compilers*. Tech. rep. Compiler Design Lab - Saarland University, 2015.

[22] Sigurd Schneider, Gert Smolka, and Sebastian Hack. *Inductive Proof Methods for Functional SSA in Verified Compilers*. Tech. rep. Compiler Design Lab - Saarland University, to appear.

[23] Michael Stepp, Ross Tate, and Sorin Lerner. "Equality-based translation validator for LLVM". In: *Computer Aided Verification*. Springer. 2011, pp. 737–742.

[24] Ross Tate et al. "Equality saturation: a new approach to optimization". In: *ACM SIGPLAN Notices*. Vol. 44. 1. ACM. 2009, pp. 264–276.

[25] *The Coq Proof Assistant*. URL: `https://coq.inria.fr`.