Proof-Producing Translation of Functional Programs into a Time & Space Reasonable Model

Kevin Kappelmann, Fabian Huch*, Lukas Stevens, Mohammad Abdulaziz

King's College London, Bush House 30 Aldwych, London WC2B 4BG, UK,
mohammad.abdulaziz@kcl.ac.uk
Technische Universität München, Boltzmannstrasse 3, Garching 85748, Germany,
kevin.kappelmann@tum.de, huch@in.tum.de, lukas.stevens@in.tum.de

Abstract. We present a semi-automated framework to construct and reason about programs in a deeply-embedded while-language. The while-language we consider is a simple computation model that can simulate (and be simulated by) Turing Machines with a linear time and constant space blow-up. Our framework derives while-programs from functional programs written in a subset of Isabelle/HOL, namely tail-recursive functions with first-order arguments and algebraic datatypes. As far as we are aware, it is the first framework targeting a computation model that is reasonable in time and space from a complexity-theoretic perspective.

Keywords: Program synthesis · Certified compilation · Interactive theorem proving · Complexity theory · Computation models

1 Introduction

A large array of mathematical results has been formlised in interactive theorem provers. One example is theoretical computer science, where extensive results on algorithm correctness and computational objects, such as reductions, were proven. Nonetheless, one type of results has proved to be particularly hard to formalise: the construction and verification of algorithms modelled within a deeply embedded computation model, such as a Turing machine. These constructions, however, are crucial to formalise the theory of efficient algorithms and computational complexity theory as understood by practitioners of theoretical computing. In these fields, most results show that an algorithm runs in polynomial time (worst-case, average case, etc.) when implemented as a program within a reasonable computation model, i.e. a model that can simulate Turing machines with a polynomial time and constant space overhead for *all* computations [**?**]. Previous work on reasoning about concrete programs within a deeply embedded computation model did so in one of two ways:

(1) Automated synthesis of deeply embedded programs from functional programs in the theorem prover's logic along with equivalence proofs [**?**, **?**]. In

* Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under the National Research Data Infrastructure – NFDI 52/1 – 501930651

the context of formalising complexity theory, this approach was only used for a lambda calculus model so far [**?**]. In practical verification contexts, it was used for a lambda calculus model [**?**] and assembly [**?**].

(2) Interactively proving properties of deeply embedded programs, mainly using Hoare or other program logics. This approach is used in many practical verification settings [**?**, **?**, **?**] and has been applied to imperative languages, including toy languages [**?**], C, Java, and Rust. In the context of formalising theoretical computer science, this approach was mainly used for Turing machines [**?**, **?**, **?**], with few applications to lambda calculus [**?**, **?**] and partial recursive functions [**?**].

To formalise the theory of efficient algorithms and complexity theory, we suggest the synthesis into a simple, imperative, and reasonable computation model.

Why? As others have noted [**?**, **?**], reasoning about programs in a theorem prover's native language is significantly easier than reasoning about deeply embedded programs. This is crucial when verifying complex algorithms. For example, Gäher and Kunze [**?**] proved the Cook-Levin theorem with native Coq programs, benefiting from existing tools and automation. In contrast, Balbach [**?**] directly reasoned about Turing machines in Isabelle/HOL, requiring extensive low-level reasoning. Others [**?**] note difficulties when reasoning about deeply-embedded programs, e.g. missing warning messages and no static type checking.

Although automated synthesis methods exist for lambda calculi, we believe it is worthwhile to develop them for an imperative model. The main reason is that the lambda calculi used in previous work are *less equivalent* to Turing machines than imperative models: they are only reasonable for computations solving decision problems but not reasonable in space in general [**?**]. This limits their applicability when analysing complexity-theoretic questions.

Finally, while there exists a synthesis-based method for x86-assembly in Coq [**?**], it does not support user-defined datatypes, only allows pattern matching and recursion with fixed recursion schemes, and its target language's semantics are vastly more complicated than needed for complexity-theoretic questions.

Contribution We present a framework that synthesises verified programs in a simple while-language with finite machine words in Isabelle/HOL. Our framework takes tail-recursive, first-order functions[1] with algebraic datatypes as input. We call this fragment HOL^{TC}. Given a HOL^{TC} function f, our proof-producing metaprogram synthesises a related HOL^{TC} function which only operates on natural numbers. We call this HOL^{TC} fragment $\text{HOL}^{\text{TC}\mathbb{N}}$.

The resulting $\text{HOL}^{\text{TC}\mathbb{N}}$ function $f^{\text{TC}\mathbb{N}}$ is compiled into a program p of our deeply embedded, imperative language IMP^{TC}, which supports function calls and tail-recursion. We provide automation to interactively prove the correspondence between $f^{\text{TC}\mathbb{N}}$ and p. In all examples we tried, the proofs were automatic. Finally, a verified chain of compilers in Isabelle/HOL translates IMP^{TC} to our target

[1] Although the framework does not support higher-order functions, each first-order instance of such a function can be compiled as described in **??**.

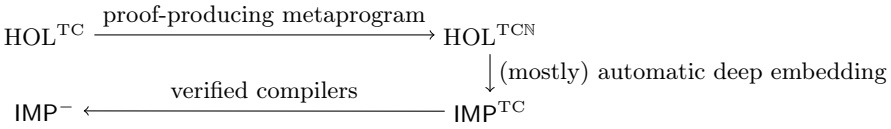

Fig. 1: Compilation pipeline from $\mathrm{HOL}^{\mathrm{TC}}$ functions to IMP^- programs.

machine language IMP^-. **??** gives an overview. The blow-up introduced by each compiler is linear in the execution time, except for the last step, where it is quadratic.

The target language only has one-bit-wide registers and only allows comparison to zero as well as bit assignments. Both operations take constant time. This is in addition to the regular program constructs of if-then-else and while-loops. Hence it is clear that IMP^- is reasonable with respect to memory and time.

Availability This article's supplementary material[2] includes the formalisation and a guide linking all definitions, results, and examples to their counterpart in the formalisation. **????????** and all underlying sub-results are formalised in Isabelle/HOL. Other proofs are provided in the appendix.

2 Preliminaries

We built our framework in Isabelle/HOL [**?**], but our approach could be followed in any simple type theory (higher-order logic) [**?**] or more expressive foundation. We use standard lambda calculi syntax, i.e. $t ::= \lambda x.\, t \mid t_1\, t_2 \mid x \mid f$, where $\lambda x.\, t$ denotes function abstractions, $t_1\, t_2$ function applications, x bound variables, and f functions with defining equations $f = \lambda \vec{x}.\, t$. We write $t : \alpha$ for "t has type α". We use vector notation for n-ary applications, abstractions, and inputs:

$$f\ \vec{x} \equiv f\, x_1 \cdots x_n \qquad \lambda \vec{x}.\, t \equiv \lambda x_1 \cdots x_n.\, t \tag{1}$$

$$\overrightarrow{f_i\, x_i} \equiv (f_1\, x_1) \cdots (f_n\, x_n) \qquad \vec{\alpha} \Rightarrow \alpha \equiv \alpha_1 \Rightarrow \cdots \alpha_n \Rightarrow \alpha \tag{2}$$

A function $f : \vec{\alpha} \Rightarrow \alpha$ is called *higher-order* if some α_i is a function type. Otherwise, it is called *first-order*. An application $f\ \vec{x} : \alpha$ is called *higher-order* if α is a function type. Otherwise, it is called *first-order*. An *algebraic datatype* (ADT) $\vec{\alpha}\, d = C_1\, \overrightarrow{\alpha_1} \mid \cdots \mid C_n\, \overrightarrow{\alpha_n}$ is a possibly recursive sum type of product types with constructors $C_i : \overrightarrow{\alpha_i} \Rightarrow \vec{\alpha}\, d$ and case combinator $\mathsf{case}_d : (\overrightarrow{\alpha_1} \Rightarrow \alpha) \Rightarrow \cdots \Rightarrow (\overrightarrow{\alpha_n} \Rightarrow \alpha) \Rightarrow \vec{\alpha}\, d \Rightarrow \alpha$.

Our framework takes as an *input language* tail-recursive, first-order functions $f = \lambda \vec{x}.\, t$, where $\vec{x}$ are the *arguments* bound in the *body* t. The body may use other such functions and ADT functions in first-order applications. **??** shows the grammar for recursive, first-order function bodies. Our input language $\mathrm{HOL}^{\mathrm{TC}}$ is the subset where each recursive call $f\ \vec{t}$ is in tail-position.

[2] swh:1:dir:0885d0653246b8b2e81f168ed4a596d29200deed

$$
\begin{array}{lr}
t ::= \mathsf{let}\, x = t_1\, \mathsf{in}\, t_2 & \text{(bind } t_1 \text{ to } x \text{ in } t_2\text{)} \\
\quad \mid (\mathsf{case}_d\, t\, \mathsf{of}\, \overrightarrow{x_1} \Rightarrow t_1 \mid \cdots \mid \overrightarrow{x_n} \Rightarrow t_n) & \text{(case combinator of ADT } \vec{\alpha}\, d\text{)} \\
\quad \mid h\ \vec{t} & \text{(first-order application)} \\
h ::= x \mid g \mid f & \text{(bound variable } x\text{, function } g\text{, or } f \text{ itself)}
\end{array}
$$

Fig. 2: Grammar for body t of a first-order function $f = \lambda \vec{x}.\, t$ with ADTs.

$$[\![\mathtt{C}\, n]\!]_s \equiv n, \quad [\![\mathtt{R}\, r]\!]_s \equiv s\, r \qquad [\![A_1 \otimes A_2]\!]_s \equiv [\![A_1]\!]_s \otimes [\![A_2]\!]_s, \text{ for } \otimes \in \{+, -\}$$

(a) Evaluation of atoms A.

(b) Evaluation rules for arithmetic expressions.

$$\textsc{Assign}\ \frac{s' = s(r := [\![a]\!]_s)}{(r \leftarrow a, s) \Rightarrow^{\mathcal{C}} s'} \qquad \textsc{IfT}\ \frac{s\, r \neq 0 \qquad (p_1, s) \Rightarrow^{n} s'}{(\mathtt{IF}\ r\ \mathtt{THEN}\ p_1\ \mathtt{ELSE}\ p_2, s) \Rightarrow^{n+\mathcal{C}} s'}$$

$$\textsc{Seq}\ \frac{(p_1, s) \Rightarrow^{n_1} s' \qquad (p_2, s') \Rightarrow^{n_2} s''}{(p_1\ ;\ p_2, s) \Rightarrow^{n_1+n_2+\mathcal{C}} s''} \qquad \textsc{IfF}\ \frac{s\, r = 0 \qquad (p_2, s) \Rightarrow^{n} s'}{(\mathtt{IF}\ r\ \mathtt{THEN}\ p_1\ \mathtt{ELSE}\ p_2, s) \Rightarrow^{n+\mathcal{C}} s'}$$

(c) Execution relation for commands, where each $\mathcal{C}$ is a fixed, non-negative constant.

Fig. 3: Semantics shared by our IMP-languages, excluding the final IMP$^-$.

We use several deeply-embedded, deterministic languages based on the *imperative language* IMP from Winskel's book [**?**], which is well-studied in formal verification [**?**, **?**]. Our IMP languages operate on a *state* of type string $\Rightarrow$ val. Terms of type string are called *registers* and terms of type val are called *values*. The registers in a program p are denoted by regs p. As a computation model, IMP allows simple reasoning about space since there is no indirect memory addressing – only registers occurring in a program can be accessed. In our final model, IMP$^-$, values are single bits, making the space usage bounded by regs p. We define IMP$^-$ in **??**, and a summary of all IMP languages can be found in **??**. Here, we describe the intermediate languages used during compilation. They use values of type $\mathbb{N}$ and consist of *atoms*, *(arithmetic) expressions* and *(imperative) commands*. Every atom is also an arithmetic expression.

We now define the *semantics*. With $[\![A]\!]_s$ and $[\![a]\!]_s$ we denote the evaluation (returning a natural number) of an atom A and expression a, respectively, under state s, defined by the rules in **????**. With $tp \vdash (p, s) \Rightarrow^n_R s'$ we denote that in context of the program tp,[3] the program p started from s terminates in at most n steps with state s'' such that $s'\, r = s''\, r$ for all $r \in R$:

$$\big(tp \vdash (p, s) \Rightarrow^n_R s'\big) \equiv \exists n'\, s''.\ tp \vdash (p, s) \Rightarrow^{n'} s'' \wedge \forall r \in R.\, s'\, r = s''\, r \wedge n' \leq n,$$

[3] The context is only relevant for recursive calls in **??**.

where $\Rightarrow^n$ is the *(big-step) execution relation*, defined inductively using the rules from **??** along with the language-specific rules specified in their respective sections. In rule **??**, the state s is updated at register r using the state update notation $s(r := v) \equiv \lambda r'.\, \mathsf{if}\ r = r'\ \mathsf{then}\ v\ \mathsf{else}\ s\, r'$. If we are only interested in a single return register r, we write

$$\big(tp \vdash (p, s) \Rightarrow_r v\big) \equiv \exists s'.\ \big(tp \vdash (p, s) \Rightarrow s' \wedge s'\, r = v\big).$$

To simplify notation, we omit "$\ldots \vdash$" if the context is irrelevant, R if the program terminates exactly with state s', and n if the number of steps is irrelevant.

One may question why rule **??** only takes constant, and not logarithmic, time. This is merely for simplicity and makes no difference for polynomial time considerations since logarithmic space usage (i.e. space usage in a binary encoding of $\mathbb{N}$) is bounded by time in all intermediate IMP languages, as shown in **??**. For this, let $a_{\max}$ denote the largest constant in an expression a, $p_{\max}$ the largest constant in a program p, and $s_{\max} \equiv \max\{s\, r \mid r : \mathsf{string}\}$.

Theorem 1. *If* $max\{s_{max}, p_{max}\} < 2^w$ *and* $(p, s) \Rightarrow^n s'$*, then* $s'_{max} < 2^{w+n}$.

Proof sketch. The proof is by induction over the execution relation. In case of an assignment, the maximum size can at most double. In other cases, the claim holds by applying the induction hypothesis, followed by algebraic manipulation. □

3 HOL$^{(\mathrm{TC})}$ to HOL$^{(\mathrm{TC})\mathbb{N}}$

This section describes the proof-producing translation of HOL$^{\mathrm{TC}}$ functions into equivalent HOL$^{\mathrm{TC}\mathbb{N}}$ functions. More generally, our translation applies to any HOL function f with ADTs, producing a HOL$^{\mathbb{N}}$ function $f^{\mathbb{N}}$ that operates only on natural numbers while preserving tail-recursiveness and first-order applications.

Intuitively, the function f and its translation $f^{\mathbb{N}}$ should compute the same values for the same inputs, up to encoding and decoding. Our method is based on a technique called *transport*, which, intuitively, takes objects of one type and derives that "corresponding" objects exist of another type, synthesising the latter objects as necessary. To formalise this correspondence, we first review some concepts on relations.

Definition 1. *A* relation on α and β *is a function of type* $\alpha \Rightarrow \beta \Rightarrow \mathsf{bool}$. *Two functions* f, g *are* related from R to S *if they map* R*-related inputs to* S*-related outputs, written* $(R \Rrightarrow S)\, f\, g \equiv \forall x\, y.\, R\, x\, y \longrightarrow S\, (f\, x)\, (g\, y)$. *Repeated relations are abbreviated as* $(R^n \Rrightarrow S) \equiv (R \Rrightarrow \cdots \Rrightarrow R \Rrightarrow S)$, *i.e.* R *repeated* n*-times.*

We can now define the desired relation between f and $f^{\mathbb{N}}$.

Definition 2. *A type* α *is* encodable *if there are functions* $\mathsf{natify} : \alpha \Rightarrow \mathbb{N}$ *and* $\mathsf{denatify} : \mathbb{N} \Rightarrow \alpha$ *such that* $\mathsf{denatify}\, (\mathsf{natify}\, x) = x$ *for all* x.[4]

[4] In the formalisation, natify, denatify are overloaded as part of a typeclass.

We say that $n : \mathbb{N}$ *and* $x : \alpha$ *are* (R$\mathbb{N}$-)related *if they represent the same value:* $\mathsf{R}\mathbb{N}\, n\, x \equiv n = \mathsf{natify}\, x$. *A number* n *is* well-encoded (with respect to α) *if* $\mathsf{R}\mathbb{N}\, n\, x$ *holds for some* $x : \alpha$. *We lift this relation to functions and say that* $f^{\mathbb{N}} : \alpha^{\mathbb{N}}$ *and* $f : \alpha$ *are* R$\mathbb{N}$-related *if* $\alpha^{\mathsf{R}\mathbb{N}}\, f^{\mathbb{N}}\, f$, *where* $\alpha^{\mathbb{N}}, \alpha^{\mathsf{R}\mathbb{N}}$ *are defined recursively:*

$$(\alpha \Rightarrow \beta)^{\mathbb{N}} \equiv \alpha^{\mathbb{N}} \Rightarrow \beta^{\mathbb{N}}, \quad \alpha^{\mathbb{N}} \equiv \mathbb{N}, \qquad (\alpha \Rightarrow \beta)^{\mathsf{R}\mathbb{N}} \equiv \alpha^{\mathsf{R}\mathbb{N}} \Rrightarrow \beta^{\mathsf{R}\mathbb{N}}, \quad \alpha^{\mathsf{R}\mathbb{N}} \equiv \mathsf{R}\mathbb{N}.$$

Objective 1. *Given a* HOL *function* f, *synthesise a* R$\mathbb{N}$*-related* HOL$^{\mathbb{N}}$ *term* $f^{\mathbb{N}}$. *Moreover, the synthesis preserves tail-recursiveness and first-order applications on well-encoded inputs.*

We achieve this in two steps: First, we encode the datatypes used by f (**??**). Second, we synthesise a HOL$^{\mathbb{N}}$ term $f^{\mathbb{N}}$ related to f (**??**).

3.1 Encoding of Datatypes

Let $\vec{\alpha}\, d = C_1\, \overrightarrow{\alpha_1} \mid \cdots \mid C_n\, \overrightarrow{\alpha_n}$ be an ADT with $a_i \equiv |\overrightarrow{\alpha_i}|$. We want to encode $\vec{\alpha}\, d$ according to **??**. Assuming that all $\alpha \in \overrightarrow{\alpha_1} \cup \cdots \cup \overrightarrow{\alpha_n}$ are encoded or $\alpha = \vec{\alpha}\, d$, the existence of suitable en- and decoding functions is straightforward. Following Gordon's approach [**?**] to encode the type of lists, each value $C_i\, \vec{x}$ is encoded as a tagged and nested pair. Datatypes are encoded recursively this way, using natural numbers as the base type:

Definition 3. *Fix an injective pairing function* $\mathsf{pair} : \mathbb{N}^2 \Rightarrow \mathbb{N}$ *with inverses* $\mathsf{fst}, \mathsf{snd}$, *i.e.* $\mathsf{fst}\,(\mathsf{pair}\, n\, m) = n$ *and* $\mathsf{snd}\,(\mathsf{pair}\, n\, m) = m$. *We define* $C_i^{\mathbb{N}} : \mathbb{N}^{a_i} \Rightarrow \mathbb{N}$ *by* $C_i^{\mathbb{N}}\, \vec{x} \equiv \mathsf{pair}\, i\, \big(\mathsf{if}\, a_i = 0\, \mathsf{then}\, 0\, \mathsf{else}\, (\mathsf{pair}\, x_1\, (\mathsf{pair}\, x_2 (\cdots (\mathsf{pair}\, x_{a_i - 1}\, x_{a_i}) \cdots))\big)$. *For each* i, j, *we define the* selector $\mathsf{select}_{i,j}\, x \equiv (\mathsf{if}\, j < i\, \mathsf{then}\, \mathsf{fst}\, \mathsf{else}\, \mathsf{id})\, (\mathsf{snd}^j\, x)$. *Using the selector, we can define a companion* $\mathsf{case}_d^{\mathbb{N}}$ *for the case combinator* case_d *using a nested if-then-else construction. Details can be found in* **??**.

These functions suffice to show that $\vec{\alpha}\, t$ is encodable:

Theorem 2. $\vec{\alpha}\, d = C_1\, \overrightarrow{\alpha_1} \mid \cdots \mid C_n\, \overrightarrow{\alpha_n}$ *is encodable if all* $\alpha \in \overrightarrow{\alpha_1} \cup \cdots \cup \overrightarrow{\alpha_n}$ *are encoded or* $\alpha = \vec{\alpha}\, d$. *Also, for all* $1 \le i \le n$, $1 \le j \le a_i$ *and relations* R, *we have*

$$(\mathsf{R}\mathbb{N}^{a_i} \Rrightarrow \mathsf{R}\mathbb{N})\, C_i^{\mathbb{N}}\, C_i, \qquad \mathsf{R}\mathbb{N}\, n\, (C_i\, \vec{x}) \implies \mathsf{R}\mathbb{N}\, (\mathsf{select}_{a_i,j} n)\, x_j,$$
$$\big((\mathsf{R}\mathbb{N}^{a_1} \Rrightarrow R) \Rrightarrow \cdots \Rrightarrow (\mathsf{R}\mathbb{N}^{a_n} \Rrightarrow R) \Rrightarrow \mathsf{R}\mathbb{N} \Rrightarrow R\big)\, \mathsf{case}_d^{\mathbb{N}}\, \mathsf{case}_d.$$

The described pipeline is fully automatic in our framework. An example application can be found in **??**. We note that our underlying pairing function is the Cantor pairing function, which results in a number computable in time polynomial in the sizes of the two numbers in the encoded pair.

3.2 Synthesis of HOL$^{\mathbb{N}}$ Functions

Let $f = \lambda \vec{x}.\, t$ be a HOL function of type α. We aim for **??**. For this, we can assume that all ADTs in α are encoded and every user-specified function $g \neq f$ in t is already compiled – and thus R$\mathbb{N}$-related to some $g^{\mathbb{N}}$.

Our approach combines *black-box and white-box transports* [**?**, **?**, **?**] in a novel way, to the best of our knowledge. Both kinds of transport take a term $t : \alpha$ and a relation R as input and synthesise a term t' together with a proof that $R\,t'\,t$. While black-box transports do so using only the structure of α and R, white-box transports also use the structure of t.

Our approach is as follows: (1) Use black-box transport to obtain a function $f^{\mathbb{N}}$ that is $\mathsf{R}\mathbb{N}$-related to f. (2) Use white-box transport to obtain a term $(\lambda\vec{x}.\,t)^{\boxdot}$ that uses $f^{\mathbb{N}}$ and which is also $\mathsf{R}\mathbb{N}$-related to f. (3) Derive the recursive equation $f^{\mathbb{N}} = (\lambda\vec{x}.\,t)^{\boxdot}$ using that $\mathsf{R}\mathbb{N}$ is left-unique. The use of black-box transports relies on the fact that $\mathsf{R}\mathbb{N}$ can be used as a relation by Transport [**?**]:

Theorem 3 (Blackbox-Transport [?]). $\mathsf{R}\mathbb{N}$, natify, *and* $\mathsf{denatify}$ *form a partial Galois equivalence. Thus, there is some* $f^{\mathbb{N}}$ *that is* $\mathsf{R}\mathbb{N}$*-related to* f.

In the formalisation, we use the Transport prototype [**?**] to obtain $f^{\mathbb{N}}$. Since the black-box transport disregards the structure of t, **??** may not preserve tail-recursiveness and first-order applications. To derive an equation $f^{\mathbb{N}} = \lambda\overrightarrow{x^{\mathbb{N}}}.\,t^{\mathbb{N}}$ structurally related to the one of f, we use white-box transport:

Definition 4. *The* white-box-transport *of* $\lambda\vec{x}.\,t$ *replaces every subterm by its* $\mathsf{R}\mathbb{N}$*-related companion. It is defined recursively by*

$$(\lambda\vec{x}.\,s)^{\boxdot} \equiv \lambda\overrightarrow{x^{\mathbb{N}}}.\,s^{\boxdot}, \qquad (t_1\,t_2)^{\boxdot} \equiv t_1^{\boxdot}\,t_2^{\boxdot}, \qquad x_i^{\boxdot} \equiv x_i^{\mathbb{N}}, \qquad g^{\boxdot} \equiv g^{\mathbb{N}}, \qquad f^{\boxdot} \equiv f^{\mathbb{N}}.$$

Theorem 4 (Whitebox-Transport [?]). $(\lambda\vec{x}.\,t)^{\boxdot}$ *and* $\lambda\vec{x}.\,t$ *are* $\mathsf{R}\mathbb{N}$*-related and white-box transports preserve tail-recursiveness and first-order applications.*

Theorem 5. *If* $\overrightarrow{\mathsf{R}\mathbb{N}\,x_i^{\mathbb{N}}\,x_i}$ *then* $f^{\mathbb{N}}\,\overrightarrow{x^{\mathbb{N}}} = t^{\boxdot}$.

Proof. We have $\mathsf{R}\mathbb{N}\,(f^{\mathbb{N}}\,\overrightarrow{x^{\mathbb{N}}})\,(f\,\vec{x})$ and $\mathsf{R}\mathbb{N}\,(t^{\boxdot})\,(f\,\vec{x})$ by **????**. Moreover, $\mathsf{R}\mathbb{N}$ is left-unique by **??**. Thus $f^{\mathbb{N}}\,\overrightarrow{x^{\mathbb{N}}} = t^{\boxdot}$. □

?????? show that we have achieved **??**. The described synthesis is fully automatic in our framework. An example is shown in **??**.

4 HOL$^{\mathrm{TC}\mathbb{N}}$ to IMP$^{\mathrm{TC}}$

We next describe the deep embedding from HOL$^{\mathrm{TC}\mathbb{N}}$ to IMP$^{\mathrm{TC}}$. **??** shows the IMP$^{\mathrm{TC}}$-specific commands and their semantics, extending those in **??**. Notably, IMP$^{\mathrm{TC}}$ supports calls of IMP$^{\mathrm{W}}$ programs (defined in **??**) and tail-recursion. It also supports if-then-elses but no case combinators. Since every $\mathsf{case}_d^{\mathbb{N}}$ is defined as a nested if-then-else construction (**??**), this is sufficient.

To relate a HOL$^{\mathrm{TC}\mathbb{N}}$ function $f^{\mathrm{TC}\mathbb{N}}$, with its conditional (recursive) equation $\overrightarrow{\mathsf{R}\mathbb{N}\,x_i^{\mathbb{N}}\,x_i} \implies f^{\mathrm{TC}\mathbb{N}}\,\overrightarrow{x^{\mathbb{N}}} = t^{\mathrm{TC}\mathbb{N}}$, to an IMP$^{\mathrm{TC}}$ program p, we presume injective

[5] There is a commmand in Isabelle/HOL that turns functions specified by multiple equations into an equivalent definition consisting of only one equation.

Listing 1.1 Input function from the user.

```
count a xs n = case xs of [] ⇒ n
| x # xs ⇒ count a xs
    (if x = a then Suc n else n)
```

Listing 1.2 Constant $\texttt{count}^{\mathbb{N}}$ and relatedness theorem from black-box transport.

```
(R𝕟 ⇛ R𝕟 ⇛ R𝕟 ⇛ R𝕟) count^𝕟 count
```

Listing 1.3 White-box transport theorem.

```
(R𝕟 ⇛ R𝕟 ⇛ R𝕟 ⇛ R𝕟)
(λa^𝕟 xs^𝕟 n^𝕟. case^𝕟 xs^𝕟 of []^𝕟 ⇒ n^𝕟
| x^𝕟 #^𝕟 xs^𝕟 ⇒ count^𝕟 a^𝕟 xs^𝕟
  (if x^𝕟 = a^𝕟 then Suc^𝕟 n^𝕟 else n^𝕟))
count
```

Listing 1.4 Final tail-recursive equation.

```
R𝕟 a^𝕟 a ∧ R𝕟 xs^𝕟 xs ∧ R𝕟 n^𝕟 n ⟹
count^𝕟 a^𝕟 xs^𝕟 n^𝕟 = case^𝕟 xs^𝕟 of
  []^𝕟 ⇒ n^𝕟
| x^𝕟 #^𝕟 xs^𝕟 ⇒ count^𝕟 a^𝕟 xs^𝕟
  (if x^𝕟 = a^𝕟 then Suc^𝕟 n^𝕟 else n^𝕟)
```

Fig. 4: For the user-specified function[5] `count` in **??**, we obtain a related constant $\texttt{count}^{\mathbb{N}}$ in **??** using **??**. Using **??**, we obtain a second term related to `count` in **??**. Using that R𝕟 is left-unique (cf. **??**), we derive the desired equation for $\texttt{count}^{\mathbb{N}}$ in **??**.

$$\textsc{Call}\ \frac{(pc, s) \Rightarrow_r^n v \qquad s' = s(r := v)}{p \vdash (\texttt{CALL}\ pc\ \texttt{RETURN}\ r, s) \Rightarrow^n s'} \qquad \textsc{Rec}\ \frac{p \vdash (p, s) \Rightarrow^n s'}{p \vdash (\texttt{RECURSE}, s) \Rightarrow^{n+c} s'}$$

Fig. 5: Execution relation of $\mathsf{IMP}^{\mathrm{TC}}$-specific commands.

functions $\mathsf{arg}, \mathsf{ret}$ such that (1) $\mathsf{arg}_{f,i}$ returns a unique name for the i-th argument of f and (2) ret_f returns a unique name for the result computed by f.

We focus on total, functional correctness of p in this work (proving concrete time bounds for p is left as future work, see **??**): From any state s with well-encoded inputs $\overrightarrow{s\ \mathsf{arg}_{f,i}}$, the program p terminates and the value in return register ret_f is equal to $f^{\mathrm{TC}\mathbb{N}}\ \overrightarrow{s\ \mathsf{arg}_{f,i}}$. Formally, our objective is:

Objective 2. *Given a* $\mathrm{HOL}^{\mathrm{TC}\mathbb{N}}$ *function* $f^{\mathrm{TC}\mathbb{N}}$*, compile it to an* $\mathsf{IMP}^{\mathrm{TC}}$ *program* p *such that* $p \vdash (p, s) \Rightarrow_{\mathsf{ret}_f} f^{\mathrm{TC}\mathbb{N}}\ \overrightarrow{s\ \mathsf{arg}_{f,i}}$ *whenever* $\overrightarrow{\mathsf{R}\mathbb{N}\ (s\ \mathsf{arg}_{f,i})\ x_i}$.

The deep embedding has two steps: (1) Compile $f^{\mathrm{TC}\mathbb{N}}$ into an $\mathsf{IMP}^{\mathrm{TC}}$ program p (**??**) and (2) prove the equivalence between $f^{\mathrm{TC}\mathbb{N}}$ and p using custom-built automation (**??**). The automation uses symbolic execution, normalises the (otherwise incomprehensible) program state such that it becomes amenable to automatic proof, and discharges well-encodedness side-conditions.

Note on Higher-Order Functions Since $\mathsf{IMP}^{\mathrm{TC}}$ does not support higher-order functions, we assume first-order functions as input for our framework. Nevertheless, each first-order instance of a higher-order function can be compiled systematically with our framework. We demonstrate this by means of an example. More cases can be found in the formalisation:

$$
\begin{aligned}
[\![\mathbf{If}\, t_1\, t_2\, t_3]\!]^b_r &\equiv [\![t_1]\!]^b_x \,;\, \texttt{IF}\ x\ \texttt{THEN}\ [\![t_2]\!]^b_r\ \texttt{ELSE}\ [\![t_3]\!]^b_r && \text{(fresh } x\text{)}\\
[\![\mathbf{Let}\, t_1\, t_2]\!]^b_r &\equiv [\![t_1]\!]^b_x \,;\, [\![t_2]\!]^{x\#b}_r && \text{(fresh } x\text{)}\\
[\![\mathbf{LetBound}\, n]\!]^b_r &\equiv r \leftarrow b\,!\,n\\
[\![\mathbf{Arg}\, n]\!]^b_r &\equiv r \leftarrow \mathsf{arg}_{f,n}\\
[\![\mathbf{Number}\, n]\!]^b_r &\equiv r \leftarrow n\\
[\![\mathbf{Call}\, g\, [t_1, \ldots, t_m]]\!]^b_r &\equiv [\![t_1]\!]^b_{x_1} \,;\ldots;\, [\![t_m]\!]^b_{x_m} \,; && \text{(fresh } x_1, \ldots, x_m\text{)}\\
&\quad\ \mathsf{arg}_{g,1} \leftarrow x_1 \,;\ldots;\, \mathsf{arg}_{g,m} \leftarrow x_m \,;\\
&\quad\ \texttt{CALL}\ g^{\mathsf{IMP}}\ \texttt{RETURN}\ \mathsf{ret}_g \,;\, r \leftarrow \mathsf{ret}_g && (g^{\mathsf{IMP}} \text{ registered for } g)\\
[\![\mathbf{Recurse}\, [t_1, \ldots, t_k]]\!]^b_r &\equiv [\![t_1]\!]^b_{x_1} \,;\ldots;\, [\![t_k]\!]^b_{x_k} && \text{(fresh } x_1, \ldots, x_k\text{)}\\
&\quad\ \mathsf{arg}_{f,1} \leftarrow x_1 \,;\ldots;\, \mathsf{arg}_{f,k} \leftarrow x_k \,;\, \texttt{RECURSE}
\end{aligned}
$$

Fig. 6: The compiler from the $\mathrm{HOL}^{\mathrm{TC}\mathbb{N}}$ representation to $\mathsf{IMP}^{\mathrm{TC}}$. Term t_1 in $\mathbf{If}\, t_1\, t_2\, t_3$ denotes the condition $t_1 \neq 0$, $\mathbf{Let}\, t_1\, t_2$ binds t_1 to the first de Bruijn index in t_2, $\mathbf{LetBound}\, i$ denotes the variable bound by the i-th enclosing **Let**, and $\mathbf{Arg}\, i$ the i-th argument of f, i.e. x_i.

Example 1. Consider the iteration function of type $(\alpha \Rightarrow \alpha) \Rightarrow \mathbb{N} \Rightarrow \alpha \Rightarrow \alpha$:

$$f^n\, x \equiv \mathsf{case}\, n\, \mathsf{of}\, 0 \Rightarrow x \mid n+1 \Rightarrow f^n\, (f\, x). \tag{3}$$

This function cannot be compiled as it is. However, every first-order instance, e.g. snd^j as used in the definition of $\mathsf{select}_{i,j}$ (**??**), can be compiled. To do so, we define $\mathsf{powsnd}\, n \equiv \mathsf{snd}^n$. We then instantiate **??** with $f = \mathsf{snd}$ and fold the definition of powsnd to obtain $\mathsf{powsnd}\, n\, x = \mathsf{case}\, n\, \mathsf{of}\, 0 \Rightarrow x \mid n+1 \Rightarrow \mathsf{powsnd}\, n\, (\mathsf{snd}\, x)$. This equation can then be compiled with our framework.

4.1 Compilation to $\mathsf{IMP}^{\mathrm{TC}}$

We compile $f^{\mathrm{TC}\mathbb{N}}\, \overrightarrow{x^{\mathbb{N}}} = t^{\mathrm{TC}\mathbb{N}}$ to an $\mathsf{IMP}^{\mathrm{TC}}$ program p in two steps. For ease of notation, we drop the superscript and just write f and t below.

First, we parse t into a metaprogram datatype resembling $\mathrm{HOL}^{\mathrm{TC}\mathbb{N}}$.[6] Second, we compile the parsed term to an $\mathsf{IMP}^{\mathrm{TC}}$ program, as shown in **??**. We denote the compilation by $[\![t]\!]^b_r$, where b is a list of registers holding values of terms bound by enclosing **Let** bindings and r is the register that will hold the program's result.

For a call $\mathbf{Call}\, g\, [t_1, \ldots, t_m]$, the compiler retrieves the registered $\mathsf{IMP}^{\mathrm{W}}$ implementation g^{IMP} of g. For the primitives equality, addition, and subtraction, manual implementations must be provided to the compiler. The compiler generates fresh register names when needed, namely to compile **Let** bindings, to store

[6] While the datatype allows recursive calls in non-tail positions, the compiler rejects such terms using a syntactic check.

$$\textsc{Update}\ \frac{[\![a]\!]_s \rightsquigarrow v}{s(r := [\![a]\!]_s) \rightsquigarrow s(r := v)} \qquad \textsc{Const}\ \frac{}{[\![\mathtt{C}\, n]\!]_s \rightsquigarrow n} \qquad \textsc{Reg}\ \frac{s\, r \rightsquigarrow v}{[\![\mathtt{R}\, r]\!]_s \rightsquigarrow v}$$

$$\textsc{Hit}\ \frac{r = r'}{s(r := v)\, r' \rightsquigarrow v} \quad \textsc{Cont}\ \frac{r \neq r' \qquad s\, r' \rightsquigarrow v'}{s(r := v)\, r' \rightsquigarrow v'} \quad \textsc{Arith}\ \frac{[\![A_1]\!]_s \rightsquigarrow v_1 \qquad [\![A_2]\!]_s \rightsquigarrow v_2}{[\![A_1 \otimes A_2]\!]_s \rightsquigarrow v_1 \otimes v_2}$$

Fig. 7: State normalisation rules.

the value of **If** conditions, and to save function arguments in temporary registers to prevent them from being overwritten. For example, in **Call** $g\,[t_1, \ldots, t_m]$, argument registers could be overwritten if another call of g appears within $t_1, \ldots, t_m$.

In summary, we parse t into a term t' of the metaprogram datatype and compile t' to the $\mathsf{IMP}^{\mathrm{TC}}$ program $p \equiv [\![t']\!]^{[]}_{\mathsf{ret}_f}$.

4.2 Correctness Proofs

Since the compiler to $\mathsf{IMP}^{\mathrm{TC}}$ is a metaprogram, its correctness cannot be proven in Isabelle/HOL. Instead, the correctness of each compiled program p must be verified individually through automation (a process sometimes called "certified extraction"), which we describe next.

First, p is normalised such that no recursive constructor (sequences and if-then-elses) appears on the left of a sequence. This simplifies the implementation, but shall not concern us any further (for details see **??**). The automation proceed as follows:

(1) If f is recursive, start an induction proof on $\vec{x}$ using the induction rule of f.
(2) Symbolically run p to completion or until a recursive call is encountered. Let s' be the end state.
(3) Run $f^{\mathrm{TC}\mathbb{N}}$ to completion with result r or until a recursive call $f^{\mathrm{TC}\mathbb{N}}\,\vec{y}$ is encountered.
(4) If both computations ran to completion, prove $s'\,\mathsf{ret}_f = r$. Otherwise, prove $\overrightarrow{s'\,\mathsf{arg}_{f,i}} = \vec{y}$ and conclude using the inductive hypothesis.

To run p, we use the execution rules from **????** (specialised to single return registers)[7] and the correctness theorems of subprograms. Each update to the $\mathsf{IMP}^{\mathrm{TC}}$ state is normalised with the rules from **??** to (1) to simplify the proof state for interactive proofs and (2) to speed up the automation. The automation eliminates cases where p and $f^{\mathrm{TC}\mathbb{N}}$ took different execution branches.

Since our correctness theorems are conditional, the inductive hypothesis and the subprograms' correctness theorems are conditional too. Whenever they are used, goals of the form $\mathsf{R}\mathbb{N}\, x^{\mathbb{N}}\, x$ must be solved. This is done automatically by a resolution-based method that uses **??**, the relatedness theorems of ADTs and

[7] The specialisation is routine and can be found in **??**.

Listing 1.5 Initial goal.

```
RN (s "a") a ∧ RN (s "xs") xs ∧ RN (s "n") n ⟹
⊢ (count^IMP, s) ⇒_ret (count^N (s "a") (s "xs") (s "n"))
```

Listing 1.6 `(#)`-case after induction setup.

```
RN (s "a") a ∧ RN (s "xs") (x # xs) ∧ RN (s "n") n ⟹ (*RN-assms.*)
(∀s'. RN (s' "a") a ∧ RN (s' "xs") xs          (*inductive hypothesis*)
  ∧ RN (s' "n") (if x = a then Suc n else n) ⟶
  ⊢ (count^IMP, s') ⇒_ret (count^N (s' "a") (s' "xs") (s' "n"))) ⟹
⊢ (count^IMP, s) ⇒_ret (count^N (s "a") (s "xs") (s "n"))
```

Listing 1.7 Subgoal after symbolic execution (repeated premises abbreviated as `<...>`).

```
<RN-assumptions> ∧ <inductive hypothesis> ∧ x = a ⟹
⊢ (count^IMP, s') ⇒_ret (count^N (s "a") (select_{2,2} (s "xs")) (Suc (s "n")))
```

Listing 1.8 Goals after application of inductive hypothesis.

```
RN (s "a") a ∧ RN (select_{2,2} (s "xs")) xs ∧ RN (Suc (s "n")) (Suc n)
```

Fig. 8: Step-by-step correctness proof for $\texttt{count}^{\mathbb{N}}$, described in **??**.

functions (**????**), and local assumptions. Note that this is essentially a form of type-checking, making sure that all involved terms are well-encoded.

Example 2. We continue with the example from **??** and show the correctness of $\texttt{count}^{\mathbb{N}}$. The goal states are shown in **??**. First, the induction rule of `count` is applied, creating cases for `[]` and `(#)`. We focus on the latter. Running $\texttt{count}^{\mathsf{IMP}}$ and $\texttt{count}^{\mathbb{N}}$ produces two subcases: one for $\texttt{x} = \texttt{a}$ and one for $\texttt{x} \neq \texttt{a}$. We consider the former. The new state `s' = s(r1 := x1,...)` is obtained from `s` by normalised state updates. Due to normalisation, we immediately get:

$$\texttt{s' "a" = s "a"} \land \texttt{s' "xs" = select}_{2,2}\texttt{(s "xs")} \land \texttt{s' "n" = Suc(s "n")}$$

Thus, the inductive hypothesis can be applied, leaving a set of relatedness goals. These are solved automatically, using the local assumptions, the selector relatedness theorems for lists, and the relatedness theorem for `Suc`.

Case Studies We applied our framework to typical functional programs, including standard datatype functions (map, fold, ...), the bisection method for square roots, and problems from complexity theory – such as reductions from SAT to 3SAT and 3SAT to independent sets. The examples can be found in the formalisation. All proofs were automatic, but functions using hundreds of registers had to be split due to inefficient retrievals for the tracked $\mathsf{IMP}^{\mathrm{TC}}$ state in the current implementation. Performance improvements are future work. We conjecture that our method is complete when applied to programs compiled from $\mathrm{HOL}^{\mathrm{TC}}$ using our framework, but a formal proof is out of scope.

$$\textsc{WhF}\ \frac{s\,r = 0}{(\texttt{WHILE}\ r\ \texttt{DO}\ p, s) \Rightarrow^{\mathcal{C}} s} \qquad \textsc{WhT}\ \frac{s\,r \neq 0 \quad (p, s) \Rightarrow^{n_1} s' \quad (\texttt{WHILE}\ r\ \texttt{DO}\ p, s') \Rightarrow^{n_2} s''}{(\texttt{WHILE}\ r\ \texttt{DO}\ p, s) \Rightarrow^{n_1+n_2+\mathcal{C}} s''}$$

Fig. 9: Execution relation for `WHILE`.

$$\frac{s\,r \neq 0 \qquad (p_1, s) \Rightarrow^{n} (s', b)}{(\texttt{IF}\ r\ \texttt{THEN}\ p_1\ \texttt{ELSE}\ p_2, s) \Rightarrow^{n+\mathcal{C}} (s', b)} \qquad \frac{(p_1, s) \Rightarrow^{n_1} (s', \texttt{0}) \qquad (p_2, s') \Rightarrow^{n_2} (s'', b)}{(p_1\ ;\ p_2, s) \Rightarrow^{n_1+n_2+\mathcal{C}} (s'', b)}$$

$$\frac{}{(\texttt{RECURSE}, s) \Rightarrow^{\mathcal{C}} (s, \texttt{1})} \qquad \frac{s' = s(r := [\![a]\!]_s)}{(r \leftarrow a, s) \Rightarrow^{\mathcal{C}} (s', \texttt{0})} \qquad \ldots$$

Fig. 10: Execution relation from **??** that halts with a flag when encountering a tail-recursive call. Omitted rules are routine and listed in **??**.

5 $\mathsf{IMP}^{\mathrm{TC}}$ to IMP^{-}

In this section, we describe the verified chain of compilers from $\mathsf{IMP}^{\mathrm{TC}}$ to IMP^{-}:

Objective 3. *For every* $\mathsf{IMP}^{\mathrm{TC}}$ *program* p*, compile an* IMP^{-} *program* p' *such that (up to encoding and decoding), for the same input, (1)* p *and* p' *compute the same output and (2) the number of steps* p' *takes is at most polynomially more than the number of steps* p *takes.*

Note that since the size of the values stored in registers is governed by **??**, if the above objective is satisfied, the memory used by the final IMP^{-} program is polynomial in the size of p and its running time.

The compilation proceeds in several steps: First, we compile recursive calls into while-loops of an intermediate language $\mathsf{IMP}^{\mathrm{WC}}$ (**??**), Second, we eliminate calls to other programs (**??**). Finally, we use bit blasting to compile into our final language IMP^{-} (**??**).

5.1 $\mathsf{IMP}^{\mathrm{TC}}$ to $\mathsf{IMP}^{\mathrm{WC}}$

In the first step, we replace tail-recursive calls by while-loops, whose semantics are given in **??**. In rule **??**, the loop-condition $r \neq \texttt{0}$ is checked *before* stepping into the `WHILE`. In contrast, tail-recursive calls occur at the *end* of the program. To bridge this gap, we prove the following rules for $\mathsf{IMP}^{\mathrm{TC}}$.

Lemma 1. *Let* $(p, s) \Rightarrow^{n} (s', b)$ *denote the modified execution relation shown in* **??** *that runs* p *but stops and flags whenever a tail-recursive call is encountered using* $b \in \{\texttt{0}, \texttt{1}\}$ *Then the following rules are admissible:*

$$\frac{(p, s) \Rightarrow^{n} (s', \texttt{0})}{tp \vdash (p, s) \Rightarrow^{n} s'} \qquad \frac{(p, s_1) \Rightarrow^{n_1} (s_2, \texttt{1}) \qquad tp \vdash (tp, s_2) \Rightarrow^{n_2} s_3}{tp \vdash (p, s_1) \Rightarrow^{n_1+n_2} s_3}$$

Note the similarity between these rules and those for **??** and **??**. These rules suggest a straightforward translation from p to an $\mathsf{IMP}^{\mathrm{WC}}$ program by using a loop that continues as long as a recursive call would have occurred:

Theorem 6. *Let* `cnt` *be a fresh register with respect to* regs p. *Let* $p[t/x]$ *denote the syntactic substitution of any occurrence of* x *by* t *in a program* p. *Define*

$$(\!|p|\!)_{\circlearrowleft} \equiv \texttt{cnt} \leftarrow \texttt{1}\ ;\ \texttt{WHILE cnt DO cnt} \leftarrow \texttt{0}\ ;\ p[(\texttt{cnt} \leftarrow \texttt{1})/\texttt{RECURSE}].$$

Then it holds that $p \vdash (p, s) \Rightarrow^n_{\mathsf{regs}\,p} s'$ *if and only if* $((\!|p|\!)_{\circlearrowleft}, s) \Rightarrow^{n+\mathcal{C}}_{\mathsf{regs}\,p} s'$.

Since we use an additional register, we only get partial equivalence of states in the correctness theorem. This is sufficient for our purposes nonetheless.

5.2 $\mathsf{IMP}^{\mathrm{WC}}$ to $\mathsf{IMP}^{\mathrm{W}}$

We next eliminate calls to $\mathsf{IMP}^{\mathrm{W}}$ programs. We simply inline every call to a program p^w by copying its used registers to fresh memory locations, executing the program there, and copying the result register back:

Definition 5. *Let* regs $p^w = \{r_1, \ldots, r_n\}$ *and* m *be a renaming to fresh registers. We define the inlining of a call to program* p^w *with return register* r *as*

$$(\!|\texttt{CALL}\ p^w\ \texttt{RETURN}\ r|\!)_\star \equiv m\,r_1 \leftarrow r_1\ ;\ \cdots\ ;\ m\,r_n \leftarrow r_n\ ;\ p^w[m\,x/x]\ ;\ r \leftarrow m\,r.$$

For an $\mathsf{IMP}^{\mathrm{WC}}$ *program* p, *we let* $(\!|p|\!)_\star$ *denote the* $\mathsf{IMP}^{\mathrm{W}}$ *program obtained from* p *by inlining all calls.*

Note that called programs must already be compiled to $\mathsf{IMP}^{\mathrm{W}}$ and hence do not contain further calls. Arguing about a program in a different memory location requires the obvious substitution lemma:

Lemma 2. $(p[(m\,x)/x], s) \Rightarrow^n s'$ *implies* $(p, s \circ m) \Rightarrow^n s' \circ m$ *for injective* m.

The inlining induces a blow-up that is dependent on the program size (but not the input), as shown below.

Theorem 7. $(p, s) \Rightarrow^n_{\mathsf{regs}\ p} s'$ *if* $((\!|p|\!)_\star, s) \Rightarrow^{f(n,p)}_{\mathsf{regs}\ p} s'$ *where* $f(n, p)$ *is linear in* $n \cdot |p|$. *Moreover, if* $((\!|p|\!)_\star, s) \Rightarrow^n_{\mathsf{regs}\ p} s'$ *then* $(p, s) \Rightarrow^n_{\mathsf{regs}\ p} s'$.

5.3 $\mathsf{IMP}^{\mathrm{W}}$ to IMP^{-}

Our programs so far operate on arbitrary-size natural numbers. In contrast, our final language IMP^{-} operates on single bits, represented as `0` and `1`. The commands of IMP^{-} and their semantics are shown in **??**. There are no arithmetic expressions, and all registers represent a single bit.

We use bit-blasting to compile $\mathsf{IMP}^{\mathrm{W}}$ programs to IMP^{-} programs. The bit-blasting is parametrised with a number w, indicating the bit-width of the compilation. Each $\mathsf{IMP}^{\mathrm{W}}$ register r is represented as $w + 1$ fresh IMP^{-} registers

$$\frac{b \in \{0,1\} \qquad s' = s(r := b)}{(r \leftarrow b, s) \Rightarrow^1 s'} \qquad \frac{s\,r \neq 0 \qquad (p_1, s) \Rightarrow^n s'}{(\texttt{IF } r \texttt{ THEN } p_1 \texttt{ ELSE } p_2, s) \Rightarrow^{n+1} s'}$$

$$\frac{(p_1, s) \Rightarrow^{n_1} s' \qquad (p_2, s') \Rightarrow^{n_2} s''}{(p_1 \,;\, p_2, s) \Rightarrow^{n_1+n_2+1} s''} \qquad \frac{s\,r = 0 \qquad (p_2, s) \Rightarrow^n s'}{(\texttt{IF } r \texttt{ THEN } p_1 \texttt{ ELSE } p_2, s) \Rightarrow^{n+1} s'}$$

$$\frac{s\,r = 0}{(\texttt{WHILE } r \texttt{ DO } p, s) \Rightarrow^1 s} \qquad \frac{s_1\,r \neq 0 \qquad (p, s_1) \Rightarrow^{n_1} s_2 \qquad (\texttt{WHILE } r \texttt{ DO } p, s_2) \Rightarrow^{n_2} s_3}{(\texttt{WHILE } r \texttt{ DO } p, s_1) \Rightarrow^{n_1+n_2+2} s_3}$$

Fig. 11: Execution relation for IMP^-.

$r_0, \ldots, r_w$: The IMP^- register r_i corresponds to the i-th bit of the $\mathsf{IMP}^{\mathrm{W}}$ register r, and the register r_0 indicates whether all bits are 0. Arithmetic expressions of $\mathsf{IMP}^{\mathrm{W}}$ are bit-blasted to IMP^- programs. We showcase the bit-blasting for addition here. The addition of x and y in register r is compiled to

$$\mathsf{add}^w\, x\, y\, r \equiv \mathsf{copy}^w\, a\, x \,;\, \mathsf{copy}^w\, b\, y \,;\, \mathsf{adder}^w\, r, \tag{4}$$

where a, b are fresh, fixed $\mathsf{IMP}^{\mathrm{W}}$ registers. The function copies the first w bits of x and y to $a_1, \ldots, a_w$ and $b_1, \ldots, b_w$, respectively, which are then added by the function $\mathsf{adder}^w\, r$. The function $\mathsf{adder}^w\, r$ puts the result of the addition in registers $r_0, \ldots, r_w$. It is implemented as follows:

$$\mathsf{adder}^w\, r \equiv \mathsf{fullAdder}^1\, r; \cdots ; \mathsf{fullAdder}^w\, r; \mathit{carry} \leftarrow 0; \mathit{zero} \leftarrow 0, \tag{5}$$

where $\mathsf{fullAdder}^i\, r$ is a program that implements a standard full adder with zero check. It adds the bits a_i and b_i and the carry register *carry*, storing the result in r_i, the new carry value in *carry*, and stores in *zero* whether the result and the previous *zero* value were 0. The full adder is called for all bits of the addends. Then r_0 is written, and the *carry* and *zero* registers are reset.

We bit-blast all other arithmetic expressions of $\mathsf{IMP}^{\mathrm{W}}$ in a similar way; details are in the formalisation. In the next definition, we fix a maximum bit-width w:

Definition 6. *We denote with $(\!|a|\!)^w_r$ the bit-blasted version of arithmetic expression a stored in register r. For an $\mathsf{IMP}^{\mathrm{W}}$ state s, we denote with $(\!|s|\!)^w$ its corresponding IMP^- state, i.e. $(\!|s|\!)^w\, r_i = (s\, r)_i$ for all registers r and $i < w$, where n_i denotes the i-th bit of n.*

The functional correctness can now be stated as follows:

Lemma 3. *If $\max\{a_{max}, [\![a]\!]_s, s_{max}\} < 2^w$ then $((\!|a|\!)^w_r, (\!|s|\!)^w) \Rightarrow^{f(w)} (\!|s(r := [\![a]\!]_s)|\!)^w$, where $f(w)$ is linear in w.*

Lastly, we show that the overall runtime blow-up of the bit-blasted program is polynomial. We first lift the bit-blasting of expressions to $\mathsf{IMP}^{\mathrm{W}}$ programs p, denoted by $(\!|p|\!)^w$. In $(\!|p|\!)^w$, arithmetic expressions are replaced with their bit-blasted

version and all other constructs are mapped to their corresponding constructs in the obvious way. For functional correctness, we get:

Theorem 8. *If* $(p, s) \Rightarrow^n s'$ *and* $n < w$ *and* $max\{s_{max}, p_{max}\} \cdot 2^n < 2^w$*, then* $(\llparenthesis p \rrparenthesis^w, \llparenthesis s \rrparenthesis^w) \Rightarrow^{f(n,w)} \llparenthesis s' \rrparenthesis^w$ *where* $f(n, w)$ *is linear in* $n \cdot w$*.*

Proof sketch. The proof is by structural induction on the IMP^- program p. There are two interesting cases. The first is the case of assignment, where the theorem follows from Lemma **??**. The other case is that of composition, in which case the theorem follows from Theorem **??**. All other cases follow from basic properties of program composition and algebraic manipulation of exponents. □

6 Discussion

Our work touches the areas of formalised computability and complexity theory, algorithm refinements and program synthesis, and certified compilation.

Notable work in the first area includes Norrish's [**?**] formalisation results, like Rice's theorem, in a lambda calculus model. Xu et al. [**?**] and Asperti and Ricciotti [**?**] proved basic results on Turing machines, using Hoare-like logics to reason about deeply-embedded machines. Carneiro [**?**] proved basic results using partial recursive functions. The richest line of work is by Forster [**?**], modelling the call-by-value lambda calculus, Turing machines, and other models, and proving simulations between them. Forster and Kunze [**?**] introduced a largely automated synthesis framework from native Coq functions to said lambda calculus model, including semi-automated time bound proofs. This was used for serious algorithms, like the Cook-Levin theorem [**?**]. However, their model is not reasonable in space in general. Our contribution is the first synthesis method into a simple computation model that is reasonable in time and space.

To make our method practical, we used ideas from algorithm refinements and transport-based program synthesis [**?**, **?**, **?**], introducing a novel synthesis for recursive functions in **??**. Other related approaches are algorithm refinements, which were deeply studied by Lammich [**?**, **?**, **?**, **?**].

In the area of certified compilation, most substantial developments geared towards practical compilers, such as CakeML [**?**] and CompCert [**?**]. We, on the other hand, we were mainly inspired by the simple and elegant use of IMP languages [**?**] in educational material [**?**, **?**], which is sufficient to formalise the theory of efficient algorithms and complexity theory.

Future Work Our framework can be extended in several ways, such as proving time and memory bounds of synthesised programs with respect to some sensible semantics for Isabelle/HOL functions [**?**], generalising from tail-recursion to general recursion, reducing the time and memory overhead of the compilers for finer analyses needed in efficient algorithms, and extending it to richer models of computation, e.g. probabilistic computation, interactive computation, or online computation. We plan to improve the framework's performance such that it can be applied to the verification of serious computational objects, e.g. reductions,

like the Cook-Levin theorem [**?**] or Karp's 21 NP-complete problems, and the verification of running time bounds of complex algorithms in general, e.g. flow algorithms [**?**, **?**] or matching algorithms [**?**]. A challenge with our framework, as in any system that synthesises programs using natural numbers as the sole datatype, is the computational overhead from encoding/decoding datatypes into natural numbers. Without special care, unsuspecting users may discrepancies between the expected and actual running time of synthesised programs. Methods to smoothly handle that discrepancy would greatly improve the usability of our framework and are a possible future direction.

Acknowledgements The authors thank Bilel Ghorbel, Florian Keßler, Max Lang, Nico Lintner, Jay Neubrand, Jonas Stahl, and Andreas Vollert for their contributions as part of their student coursework at TU Munich.

A HOL$^{(\mathbf{TC})}$ to HOL$^{(\mathbf{TC})\mathbb{N}}$

A.1 Compilation of Datatypes

Let $\vec{\alpha}\, t = C_1\, \overrightarrow{\alpha_1} \mid \cdots \mid C_n\, \overrightarrow{\alpha_n}$ be an algebraic datatype with $a_i \equiv |\overrightarrow{\alpha_i}|$. Recall that we require an injective pairing function $\mathsf{pair} : \mathbb{N}^2 \Rightarrow \mathbb{N}$ and functions $\mathsf{fst}, \mathsf{snd} : \mathbb{N} \Rightarrow \mathbb{N}$ with

$$\mathsf{fst}\,(\mathsf{pair}\, n\, m) = n \qquad \text{and} \qquad \mathsf{snd}\,(\mathsf{pair}\, n\, m) = m. \tag{6}$$

Our construction proceeds as follows:

(a) We define $C_i^{\mathbb{N}} : \mathbb{N}^{a_i} \Rightarrow \mathbb{N}$ by

$$C_i^{\mathbb{N}}\, \vec{x} \equiv \begin{cases} \mathsf{pair}\, i\, (\mathsf{pair}\, x_1\, (\mathsf{pair}\, x_2 (\cdots (\mathsf{pair}\, x_{a_i - 1}\, x_{a_i}) \cdots), & \text{if } a_i > 0 \\ \mathsf{pair}\, i\, 0, & \text{otherwise} \end{cases}. \tag{7}$$

(b) For each i, j, we define the *selector* $\mathsf{select}_{i,j} : \mathbb{N} \Rightarrow \mathbb{N}$ as

$$\mathsf{select}_{i,j}\, x \equiv (\mathsf{if}\ j < i\ \mathsf{then}\ \mathsf{fst}\ \mathsf{else}\ \mathsf{id})\, (\mathsf{snd}^j\, x). \tag{8}$$

(c) We define $\mathsf{case}^{\mathbb{N}} : \big((\mathbb{N}^{a_1} \Rightarrow \alpha) \Rightarrow \cdots \Rightarrow (\mathbb{N}^{a_n} \Rightarrow \alpha) \Rightarrow \mathbb{N} \Rightarrow \alpha\big)$ by

$$\mathsf{case}^{\mathbb{N}}\, \vec{f}\, x \equiv \begin{cases} f_i\, (\mathsf{select}_{a_i,1}\, x) \cdots (\mathsf{select}_{a_i,a_i}\, x), & \text{if } i = \mathsf{fst}\, x, 1 \leq i \leq n-1 \\ f_n\, (\mathsf{select}_{a_n,1}\, x) \cdots (\mathsf{select}_{a_n,a_n}\, x), & \text{otherwise.} \end{cases} \tag{9}$$

(d) We define $\mathsf{natify} : \vec{\alpha}\, t \Rightarrow \mathbb{N}$ and $\mathsf{denatify} : \mathbb{N} \Rightarrow \vec{\alpha}\, t$ by

$$\mathsf{natify}\, (C_i\, \vec{x}) \equiv C_i^{\mathbb{N}}\, \overrightarrow{\mathsf{natify}\, x_j}, \tag{10}$$

$$\mathsf{denatify}\, (C_i^{\mathbb{N}}\, \vec{x}) \equiv C_i\, \overrightarrow{\mathsf{denatify}\, x_j}. \tag{11}$$

Note that $\mathsf{natify}, \mathsf{denatify}$ are overloaded and **????** well-defined if all $\alpha \in \overrightarrow{\alpha_1} \cup \cdots \cup \overrightarrow{\alpha_n}$ are already encoded or $\alpha = \vec{\alpha}\, t$.

Theorem. *$\vec{\alpha}\, d = C_1\, \overrightarrow{\alpha_1} \mid \cdots \mid C_n\, \overrightarrow{\alpha_n}$ is encodable if all $\alpha \in \overrightarrow{\alpha_1} \cup \cdots \cup \overrightarrow{\alpha_n}$ are encoded or $\alpha = \vec{\alpha}\, d$. Also, for all $1 \leq i \leq n$, $1 \leq j \leq a_i$ and relations R, we have*

$$(\mathsf{R}\mathbb{N}^{a_i} \Rrightarrow \mathsf{R}\mathbb{N})\, C_i^{\mathbb{N}}\, C_i, \qquad \mathsf{R}\mathbb{N}\, n\, (C_i\, \vec{x}) \implies \mathsf{R}\mathbb{N}\, (\mathsf{select}_{a_i,j} n)\, x_j,$$
$$\big((\mathsf{R}\mathbb{N}^{a_1} \Rrightarrow R) \Rrightarrow \cdots \Rrightarrow (\mathsf{R}\mathbb{N}^{a_n} \Rrightarrow R) \Rrightarrow \mathsf{R}\mathbb{N} \Rrightarrow R\big)\, \mathsf{case}_d^{\mathbb{N}}\, \mathsf{case}_d.$$

Proof. We have to show that $\mathsf{denatify}\, (\mathsf{natify}\, x) = x$ for all x. This follows directly by definition of $\mathsf{denatify}, \mathsf{natify}$ and the fact that $\mathsf{fst}, \mathsf{snd}$ are inverse to pair and all $\alpha \in \overrightarrow{\alpha_1} \cup \cdots \cup \overrightarrow{\alpha_n}$ are encoded (and hence have inverse functions $\mathsf{natify}, \mathsf{denatify}$).

The relatedness statements follow similarly by definition and the relatedness properties of the type arguments en- and decoding functions. □

Example 3. For the type of lists, `'a list = Nil | Cons 'a "'a list"`, the following constants are defined

```
definition Nil_nat = pair_nat 1 0
definition Cons_nat x xs = pair_nat 2 (pair_nat x xs)
definition case_list_nat f1 f2 n = if fst n = 1 then f1
  else f2 (select_{2,1} n) (select_{2,2} n)
definition denatify =
  case_list_nat Nil (λx xs. Cons (denatify x) (denatify xs))
definition natify =
  case_list Nil_nat (λx xs. Cons_nat (natify x) (natify xs))
```

and the following theorems are proven:

```
lemma "R𝕹 Nil_nat Nil"
lemma "(R𝕹 ⇛ R𝕹 ⇛ R𝕹) Cons_nat Cons"
lemma "(R ⇛ (R𝕹 ⇛ R𝕹 ⇛ R) ⇛ R𝕹 ⇛ R)
  case_list_nat case_list"
lemma "R𝕹 ns (x # xs) ⟹ R𝕹 (select_{2,1} ns) x"
lemma "R𝕹 ns (x # xs) ⟹ R𝕹 (select_{2,2} ns) xs"
```

A.2 Synthesis of HOL$^{\mathbb{N}}$ Functions

Theorem. $\mathsf{R}\mathbb{N}$, natify, *and* $\mathsf{denatify}$ *form a partial Galois equivalence. Thus, there is some* $f^{\mathbb{N}}$ *that is* $\mathsf{R}\mathbb{N}$*-related to* f*.*

Proof. Due to [**?**, Lemma 2], it suffices to show that (1) $\mathsf{R}\mathbb{N}$ is right-total and right-unique, (2) $\mathsf{R}\mathbb{N}\,n\,x$ implies $\mathsf{denatify}\,n = x$, and (3) $\mathsf{R}\mathbb{N}\,(\mathsf{natify}\,x)\,x$ for all n, x. By **??**, $\mathsf{R}\mathbb{N}$ is right-total and $\mathsf{R}\mathbb{N}\,(\mathsf{natify}\,x)\,x$. Since natify is injective (**??**), $\mathsf{R}\mathbb{N}$ is also right-unique and $\mathsf{R}\mathbb{N}\,n\,x$ implies $\mathsf{denatify}\,n = x$.

The existence of $f^{\mathbb{N}}$ now follows from [**?**, Theorem 1 and 2]. □

Theorem (Whitebox-Transport [**?**])**.** $(\lambda\vec{x}.\,t)^{\boxdot}$ *and* $\lambda\vec{x}.\,t$ *are* $\mathsf{R}\mathbb{N}$*-related and white-box transports preserve tail-recursiveness and first-order applications.*

Proof. The former is shown by structural induction on t, using that f is $\mathsf{R}\mathbb{N}$-related to $f^{\mathbb{N}}$ by **??** and all other functions g are $\mathsf{R}\mathbb{N}$-related to $g^{\mathbb{N}}$ by assumption.

The latter follows immediately from **??** and structural induction on t. □

B HOL$^{\mathbb{TCN}}$ to IMP$^{\mathbf{TC}}$

B.1 Compilation to IMP$^{\mathbf{TC}}$

?? shows the datatype representation of HOL$^{\mathbb{TCN}}$ as used in the compiler to IMP$^{\mathrm{TC}}$, shown in **??**.

$$
\begin{array}{rl}
t ::= \mathbf{If}\, t_1\, t_2\, t_3 & \text{(if-then-else with natural number condition } t_1\text{)} \\
\mid \mathbf{Let}\, t_1\, t_2 & \text{(bind } t_1 \text{ to the first de Bruijn index in } t_2\text{)} \\
\mid \mathbf{LetBound}\, i & \text{(variable bound by } i\text{-th enclosing } \mathbf{Let}\text{)} \\
\mid \mathbf{Arg}\, i & \text{(}i\text{-th argument of } f\text{, i.e. } x_i\text{)} \\
\mid \mathbf{Number}\, n & \text{(natural number } n \in \mathbb{N}\text{)} \\
\mid \mathbf{Call}\, g\, [t_1, \ldots, t_m] & \text{(call to } g : \mathbb{N}^m \Rightarrow \mathbb{N} \text{ with arguments } t_1, \ldots, t_m\text{)} \\
\mid \mathbf{Recurse}\, [t_1, \ldots, t_k] & \text{(recursion with arguments } t_1, \ldots, t_k\text{)}
\end{array}
$$

Fig. 12: Datatype representation of $\mathrm{HOL}^{\mathrm{TC}\mathbb{N}}$ in the metaprogramming language.

$$
\begin{array}{rll}
[\![\mathbf{If}\, t_1\, t_2\, t_3]\!]^b_r & \equiv [\![t_1]\!]^b_x \,;\, \texttt{IF}\ x\ \texttt{THEN}\ [\![t_2]\!]^b_r\ \texttt{ELSE}\ [\![t_3]\!]^b_r & \text{(fresh } x\text{)} \\
[\![\mathbf{Let}\, t_1\, t_2]\!]^b_r & \equiv [\![t_1]\!]^b_x \,;\, [\![t_2]\!]^{x\#b}_r & \text{(fresh } x\text{)} \\
[\![\mathbf{LetBound}\, n]\!]^b_r & \equiv r \leftarrow b\,!\,n & \\
[\![\mathbf{Arg}\, n]\!]^b_r & \equiv r \leftarrow \mathsf{arg}_{f,n} & \\
[\![\mathbf{Number}\, n]\!]^b_r & \equiv r \leftarrow n & \\
[\![\mathbf{Call}\, g\, [t_1, \ldots, t_m]]\!]^b_r & \equiv [\![t_1]\!]^b_{x_1} \,;\, \ldots \,;\, [\![t_m]\!]^b_{x_m} \,; & \text{(fresh } x_1, \ldots, x_m\text{)} \\
 & \quad \mathsf{arg}_{g,1} \leftarrow x_1 \,;\, \ldots \,;\, \mathsf{arg}_{g,m} \leftarrow x_m \,; & \\
 & \quad \texttt{CALL}\ g^{\mathsf{IMP}}\ \texttt{RETURN}\ \mathsf{ret}_g \,;\, r \leftarrow \mathsf{ret}_g & (g^{\mathsf{IMP}} \text{ registered for } g) \\
[\![\mathbf{Recurse}\, [t_1, \ldots, t_k]]\!]^b_r & \equiv [\![t_1]\!]^b_{x_1} \,;\, \ldots \,;\, [\![t_k]\!]^b_{x_k} & \text{(fresh } x_1, \ldots, x_k\text{)} \\
 & \quad \mathsf{arg}_{f,1} \leftarrow x_1 \,;\, \ldots \,;\, \mathsf{arg}_{f,k} \leftarrow x_k \,; & \\
 & \quad \texttt{RECURSE} &
\end{array}
$$

Fig. 13: The compiler from the $\mathrm{HOL}^{\mathrm{TC}\mathbb{N}}$ representation in **??** to $\mathsf{IMP}^{\mathrm{TC}}$.

B.2 Correctness Proofs

Our goal is to show that $p \vdash (p, s) \Rightarrow_{\mathsf{ret}_f} f^{\mathrm{TC}\mathbb{N}}\, \overrightarrow{s\, \mathsf{arg}_{f,i}}$, i.e. we are interested in the single return register ret_f. **??** shows the execution relation for $\mathsf{IMP}^{\mathrm{TC}}$ specialised to single return registers. Note that for sequences $(p_1\,;\,p_2, s) \Rightarrow_r v$, the standard execution relation "$\Rightarrow$" (and not "$\Rightarrow_r$") must be used for p_1, that is $(p_1, s) \Rightarrow s'$.

This is the reason why each compiled $\mathsf{IMP}^{\mathrm{TC}}$ program p is normalised such that no recursive constructor (sequences and if-then-elses) appears on the left of a sequence. This way, all goals involving recursive constructors use the execution relation "$\Rightarrow_r$", saving us from implementing separate recursive automation for "$\Rightarrow$".

$$\textsc{Assign}_1\ \frac{s(r := [\![a]\!]_s)\, r' = v}{(r \leftarrow a, s) \Rightarrow_{r'} v} \qquad \textsc{IfT}_1\ \frac{s\, r \neq 0 \qquad (p_1, s) \Rightarrow_r v}{(\texttt{IF}\ r\ \texttt{THEN}\ p_1\ \texttt{ELSE}\ p_2, s) \Rightarrow_r v}$$

$$\textsc{Seq}_1\ \frac{(p_1, s) \Rightarrow s' \qquad (p_2, s') \Rightarrow_r v}{(p_1\ ;\ p_2, s) \Rightarrow_r v} \qquad \textsc{IfF}_1\ \frac{s\, r = 0 \qquad (p_2, s) \Rightarrow_r v}{(\texttt{IF}\ r\ \texttt{THEN}\ p_1\ \texttt{ELSE}\ p_2, s) \Rightarrow_r v}$$

$$\textsc{Call}_1\ \frac{(pc, s) \Rightarrow_r v \qquad s(r := v)\, r' = v'}{(\texttt{CALL}\ pc\ \texttt{RETURN}\ r, s) \Rightarrow_{r'} v'} \qquad \textsc{Rec}_1\ \frac{p \vdash (p, s) \Rightarrow_r v}{p \vdash (\texttt{RECURSE}, s) \Rightarrow_r v}$$

Fig. 14: Execution relation of $\mathsf{IMP}^{\mathrm{TC}}$ for single return registers.

$$\frac{s' = s(r := [\![a]\!]_s)}{(r \leftarrow a, s) \Rightarrow^{\mathcal{C}} (s', \texttt{0})} \qquad \frac{s\, r \neq 0 \qquad (p_1, s) \Rightarrow^n (s', b)}{(\texttt{IF}\ r\ \texttt{THEN}\ p_1\ \texttt{ELSE}\ p_2, s) \Rightarrow^{n+\mathcal{C}} (s', b)}$$

$$\frac{(p_1, s) \Rightarrow^{n_1} (s', \texttt{0}) \qquad (p_2, s') \Rightarrow^{n_2} (s'', b)}{(p_1\ ;\ p_2, s) \Rightarrow^{n_1+n_2+\mathcal{C}} (s'', b)} \qquad \frac{s\, r = 0 \qquad (p_2, s) \Rightarrow^n (s', b)}{(\texttt{IF}\ r\ \texttt{THEN}\ p_1\ \texttt{ELSE}\ p_2, s) \Rightarrow^{n+\mathcal{C}} (s', b)}$$

$$\frac{(pc, s) \Rightarrow^n_r v \qquad s' = s(r := v)}{(\texttt{CALL}\ pc\ \texttt{RETURN}\ r, s) \Rightarrow^{n+\mathcal{C}} (s', \texttt{0})} \qquad \frac{}{(\texttt{RECURSE}, s) \Rightarrow^{\mathcal{C}} (s, \texttt{1})}$$

Fig. 15: Auxiliary execution relation used in **??**.

C $\mathsf{IMP}^{\mathrm{TC}}$ to IMP^{-}

C.1 $\mathsf{IMP}^{\mathrm{TC}}$ to $\mathsf{IMP}^{\mathrm{WC}}$

Lemma. *Let $(p, s) \Rightarrow^n (s', b)$ denote the modified execution relation shown in* **??** *that runs p but stops and flags whenever a tail-recursive call is encountered using $b \in \{\texttt{0}, \texttt{1}\}$ Then the following rules are admissible:*

$$\frac{(p, s) \Rightarrow^n (s', \texttt{0})}{(p, s) \Rightarrow^n s'} \qquad \frac{(p, s_1) \Rightarrow^{n_1} (s_2, \texttt{1}) \qquad tp \vdash (tp, s_2) \Rightarrow^{n_2} s_3}{tp \vdash (p, s_1) \Rightarrow^{n_1+n_2} s_3}$$

Proof sketch. By induction over the execution rules. □

Theorem. *Let* cnt *be a fresh register with respect to* regs p. *Let $p[t/x]$ denote the syntactic substitution of any occurrence of x by t in a program p. Define*

$$(\!|p|\!)_{\circlearrowleft} \equiv \texttt{cnt} \leftarrow \texttt{1}\ ;\ \texttt{WHILE cnt DO cnt} \leftarrow \texttt{0}\ ;\ p[(\texttt{cnt} \leftarrow \texttt{1})/\texttt{RECURSE}].$$

Then it holds that $p \vdash (p, s) \Rightarrow^n_{\mathsf{regs}\, p} s'$ if and only if $((\!|p|\!)_{\circlearrowleft}, s) \Rightarrow^{n+\mathcal{C}}_{\mathsf{regs}\, p} s'$.

Proof sketch. We show that $\vdash (p, s) \Rightarrow^n (s', b)$ holds if and only if

$$\exists s''.\, (p[(\texttt{cnt} \leftarrow \texttt{1})/\texttt{RECURSE}], s) \Rightarrow^n s'' \wedge s''\, \texttt{cnt} = b \wedge \forall r \in \mathsf{regs}\, p.\, s'\, p = s''\, p$$

by induction over the execution rules, generalising over S such that $\mathsf{regs}\, p \subseteq S$ and $\mathtt{cnt} \notin S$. Then the semantics for the while-loop with rules from **??** are equivalent to the semantics of the compiled program with rules from **??** by straightforward induction over the respective rules. □

C.2 $\mathsf{IMP}^{\mathrm{WC}}$ to $\mathsf{IMP}^{\mathrm{W}}$

Lemma. *$(p[m\,x/x], s) \Rightarrow^n s'$ implies $(p, s \circ m) \Rightarrow^n s' \circ m$ for injective m.*

Proof sketch. By induction over the execution of p. □

The inlining induces a blow-up that is dependent on the program size (but not the input), as shown below.

Theorem. *$(p, s) \Rightarrow^n_{\mathsf{regs}\ p} s'$ if and only if $(\!(p)\!)_\star, s) \Rightarrow^{f(n)}_{\mathsf{regs}\ p} s'$, where $f(n)$ is linear in $n \cdot |p|$.*

Proof sketch. The inlining is correct by **??**, so correctness follows by a simple induction. □

C.3 $\mathsf{IMP}^{\mathrm{W}}$ to IMP^{-}

Lemma. *Assume that $max\{a_{max}, [\![a]\!]_s, s_{max}\} < 2^w$. Then it holds that $((\!|a|\!)^w_r, (\!|s|\!)^w) \Rightarrow^{f(w)} (\!|s(r := [\![a]\!]_s)|\!)^w$, where $f(w)$ is linear in w.*

Proof Sketch. The proof is based on showing an upper bound on the running time of each of the functions $\mathsf{copy}^a\, b\, y$ and $\mathsf{adder}^w\, v$, which in turn depends on the running time of $\mathsf{fullAdder}^i\, r$. The running time of $\mathsf{fullAdder}^i\, r$ is a constant. The running times of adder and copy are shown to be linear in w by induction on w. □

D IMP-Definitions

?? shows the complete specification of $\mathsf{IMP}^{\mathrm{TC}}$, **??** the rule for `WHILE`, and **??** the complete specification of IMP^{-}.

$$[\![\mathtt{C}\,n]\!]_s \equiv n, \quad [\![\mathtt{R}\,r]\!]_s \equiv s\,r \qquad\qquad [\![A_1 \otimes A_2]\!]_s \equiv [\![A_1]\!]_s \otimes [\![A_2]\!]_s, \text{ for } \otimes \in \{+, -\}$$

(a) Evaluation of atoms A.

(b) Evaluation rules for arithmetic expressions.

$$\textsc{Assign}\ \frac{s' = s(r \coloneqq [\![a]\!]_s)}{(r \leftarrow a, s) \Rightarrow^{\mathcal{C}} s'} \qquad \textsc{IfT}\ \frac{s\,r \neq 0 \qquad (p_1, s) \Rightarrow^{n} s'}{(\mathtt{IF}\ r\ \mathtt{THEN}\ p_1\ \mathtt{ELSE}\ p_2, s) \Rightarrow^{n+\mathcal{C}} s'}$$

$$\textsc{Seq}\ \frac{(p_1, s) \Rightarrow^{n_1} s' \qquad (p_2, s') \Rightarrow^{n_2} s''}{(p_1\ ;\ p_2, s) \Rightarrow^{n_1+n_2+\mathcal{C}} s''} \qquad \textsc{IfF}\ \frac{s\,r = 0 \qquad (p_2, s) \Rightarrow^{n} s'}{(\mathtt{IF}\ r\ \mathtt{THEN}\ p_1\ \mathtt{ELSE}\ p_2, s) \Rightarrow^{n+\mathcal{C}} s'}$$

$$\textsc{Call}\ \frac{(pc, s) \Rightarrow^{n}_{r} v \qquad s' = s(r \coloneqq v)}{(\mathtt{CALL}\ pc\ \mathtt{RETURN}\ r, s) \Rightarrow^{n} s'} \qquad \textsc{Rec}\ \frac{p \vdash (p, s) \Rightarrow^{n} s'}{p \vdash (\mathtt{RECURSE}, s) \Rightarrow^{n+\mathcal{C}} s'}$$

(c) Execution relation for commands.

Fig. 16: Semantics of $\mathsf{IMP}^{\mathrm{TC}}$.

$$\textsc{WhF}\ \frac{s\,r = 0}{(\mathtt{WHILE}\ r\ \mathtt{DO}\ p, s) \Rightarrow^{\mathcal{C}} s}$$

$$\textsc{WhT}\ \frac{s_1\,r \neq 0 \qquad (p, s_1) \Rightarrow^{n_1} s_2 \qquad (\mathtt{WHILE}\ r\ \mathtt{DO}\ p, s_2) \Rightarrow^{n_2} s_3}{(\mathtt{WHILE}\ r\ \mathtt{DO}\ p, s_1) \Rightarrow^{n_1+n_2+\mathcal{C}} s_3}$$

Fig. 17: Execution relation for `WHILE` from $\mathsf{IMP}^{\mathrm{WC}}$ and $\mathsf{IMP}^{\mathrm{W}}$.

$$\frac{b \in \{\mathtt{0}, \mathtt{1}\} \qquad s' = s(r \coloneqq b)}{(r \leftarrow b, s) \Rightarrow^{1} s'} \qquad \frac{s\,r \neq \mathtt{0} \qquad (p_1, s) \Rightarrow^{n} s'}{(\mathtt{IF}\ r\ \mathtt{THEN}\ p_1\ \mathtt{ELSE}\ p_2, s) \Rightarrow^{n+1} s'}$$

$$\frac{(p_1, s) \Rightarrow^{n_1} s' \qquad (p_2, s') \Rightarrow^{n_2} s''}{(p_1\ ;\ p_2, s) \Rightarrow^{n_1+n_2+1} s''} \qquad \frac{s\,r = \mathtt{0} \qquad (p_2, s) \Rightarrow^{n} s'}{(\mathtt{IF}\ r\ \mathtt{THEN}\ p_1\ \mathtt{ELSE}\ p_2, s) \Rightarrow^{n+1} s'}$$

$$\frac{s\,r = \mathtt{0}}{(\mathtt{WHILE}\ r\ \mathtt{DO}\ p, s) \Rightarrow^{1} s} \qquad \frac{s_1\,r \neq \mathtt{0} \qquad (p, s_1) \Rightarrow^{n_1} s_2 \qquad (\mathtt{WHILE}\ r\ \mathtt{DO}\ p, s_2) \Rightarrow^{n_2} s_3}{(\mathtt{WHILE}\ r\ \mathtt{DO}\ p, s_1) \Rightarrow^{n_1+n_2+2} s_3}$$

Fig. 18: Execution relation for IMP^{-}.