Formal Program Optimization in Nuprl Using Computational Equivalence and Partial Types

Vincent Rahli, Mark Bickford, and Abhishek Anand

Cornell University, Ithaca, NY, USA

Abstract. This paper extends the proof methods used by the Nuprl proof assistant to reason about the computational behavior of its untyped programs. We have implemented new methods to prove non-trivial bisimulations between programs and have successfully applied these methods to formally optimize distributed programs such as our synthesized and verified version of Paxos, a widely used protocol to achieve software based replication. We prove new results about the basic computational equality relation on terms, and we extend the theory of partial types as the basis for stating internal results about the computation system that were previously treated only in the meta theory of Nuprl. All the lemmas presented in this paper have been formally proved in Nuprl.

1 Introduction

This paper presents proof techniques implemented in the Nuprl proof assistant [16,27,4] to reason about its own computation system and programming language, an applied lazy (call-by-name) λ-calculus. Since the computation system is universal (Turing complete), we need to reason using *partial types* introduced by Constable and Smith [35,17] and extended by Crary [18].[1] The *bisimulation* relation defined by Howe turned out to form a contextual equivalence relation [23,24], and is therefore the basic computational equality on Nuprl terms. Internally it becomes the equality on the partial type Base of all untyped Nuprl terms, both programs and data. The canonical values of this type are the terminating terms, the values of the type system.

Nuprl's logic is defined on top of this computation system. It is an extensional Constructive Type Theory (CTT) [16] which relies on ternary partial equivalence relations that express when two terms are equal in a type. For example, the type $1+1 =_{\mathbb{N}} 2$ expresses that $1+1$ and 2 are equal natural numbers (we write $x \in T$, for $x =_T x$). Each type is defined by such a relation.

Over the past two decades much progress has been made to enrich Nuprl and make it a practical programming language as well as a logical system in which one can verify properties of Nuprl programs [35,17,18,21,25,26]. During that period, Nuprl's theory was extended with, e.g., intersection types, union types, partial types, a call-by-value operator, rules to reason about computation, and in particular rules about the fixpoint operator. Recently, we have extended Nuprl

[1] Crary gave a denotational semantics for an ML dialect using partial types.

with new operators called *canonical form tests* (similar to Lisp and Scheme's type predicates) so that programs can distinguish between primitive canonical operators such as the pair or lambda constructors, and we have developed new ways to reason about these new constructs. This gives us more tools to program in Nuprl and reason about these programs.

Nuprl's intersection and partial types add expressive power. They allow us to reason about a larger class of practical programs and express more program properties. However, using typed equivalences to transform programs can be unnecessarily complex because programs are not annotated with types and both type checking and type inference are undecidable in Nuprl. Instead, we can reason about untyped program equivalences (e.g., between partial functions), which are easier to use because they only require trivial type reasoning. Such equivalences are highly useful for program transformation such as program optimization.

Using untyped reasoning, we have proved many bisimulations involving data structures such as lists. We have also used these techniques in our work on process synthesis [11,33], where processes are defined as recursive functions of a co-recursive type. Our synthesized processes were initially too slow to be used in industrial strength systems. In response to that issue, we have developed a proof methodology to simplify and optimize them. We have applied that methodology to various synthesized consensus protocols such as 2/3-consensus [14] or Multi-Paxos [28], and observed a significant speed-up. These synthesized consensus protocols have successfully been used in a replicated database [34]. This paper illustrates these proof techniques using a simple running example: appending the empty list to a term. It then illustrates their use to optimize distributed processes synthesized from protocol specifications.

Finally, being able to reason about Nuprl's programming language directly in Nuprl is another step towards a longstanding goal of building a correct-by-construction, workable logical programming environment [22]. An obvious question is then, could we build a verified compiler for Nuprl in Nuprl that generates reasonably fast code? Modern proof assistants that implement constructive type theories such as Coq [9,1], Isabelle [8,7], or Nuprl rely on unverified compilers. Even though the programs they generate, e.g., by extraction from proofs, are *correct-by-construction*, one could argue whether the machine code obtained after compilation is still correct. Thus, we would like these proof assistants to be expressive enough to program and verify optimized compilers for their underlying programming languages, and to program these proof assistants in themselves.

The contributions of this paper are as follows: (1) we introduce new formal untyped reasoning techniques for proving bisimulations, which expose more of the computation system to formal reasoning; and (2) we apply these techniques to optimize distributed processes.

2 Nuprl's Programming Language

2.1 Syntax

Nuprl is defined on top of an applied lazy untyped λ-calculus. Fig. 1 introduces a subset of this language, where $\underline{n}$ ranges over integers. Because this language

$v ::= \underline{n}$	(integer)	$\mid \lambda x.t$	(lambda)	
$\mid \langle t_1, t_2 \rangle$	(pair)	$\mid \texttt{inl}(t)$	(left injection)	
$\mid \texttt{inr}(t)$	(right injection)	$\mid \texttt{Ax}$	(axiom)	
$t ::= x$	(variable)	$\mid \texttt{isaxiom}(\boxed{t_1}, t_2, t_3)$	(isaxiom)	
$\mid v$	(value)	$\mid \texttt{ispair}(\boxed{t_1}, t_2, t_3)$	(ispair)	
$\mid \boxed{t_1}\ t_2$	(application)	$\mid \texttt{islambda}(\boxed{t_1}, t_2, t_3)$	(islambda)	
$\mid \texttt{fix}(\boxed{t})$	(fixpoint)	$\mid \texttt{isinl}(\boxed{t_1}, t_2, t_3)$	(isinl)	
$\mid \texttt{let}\ x := \boxed{t_1}\ \texttt{in}\ t_2$	(call-by-value)	$\mid \texttt{isinr}(\boxed{t_1}, t_2, t_3)$	(isinr)	
$\mid \texttt{let}\ x ::= \boxed{t_1}\ \texttt{in}\ t_2$	(call-by-valueall)	$\mid \texttt{isint}(\boxed{t_1}, t_2, t_3)$	(isint)	
$\mid \texttt{let}\ x, y = \boxed{t_1}\ \texttt{in}\ t_2$	(spread)			
$\mid \texttt{case}\ \boxed{t_1}\ \texttt{of inl}(x) \Rightarrow t_2 \mid \texttt{inr}(y) \Rightarrow t_3$	(decide)			
$\mid \texttt{if}\ \boxed{t_1} = \boxed{t_2}\ \texttt{then}\ t_3\ \texttt{else}\ t_4$	(integer equality)			

Fig. 1: Syntax of Nuprl's programming language

is lazy, its values[2] (or canonical forms) are either integers, lambda abstractions, pairs, injections, or `Ax`. The canonical form `Ax` (sometimes written as $\star$) is the unique canonical inhabitant of true propositions that do not have any nontrivial computational meaning in CTT, such as $0 =_{\mathbb{N}} 0$ which is an axiom of the logic. Non-canonical terms (non-values) have arguments that are said to be *principal*. These principal arguments indicate which subterms of a non-canonical term have to be evaluated before checking whether the term itself is a redex or not. Principal arguments of terms are marked with boxes in the above table. In the rest of this paper, variables will be obvious from the context (we often use x and y such as in Fig. 1), we use v for values, and the other letters can be any term. When it is more readable we write $t_1(t_2)$ instead of $t_1\ t_2$.

As mentioned above, we have recently added new primitive operators to Nuprl: the canonical form tests such as `ispair`. Adding these primitive forms was a design decision we made to distinguish between canonical forms (e.g., see `list_ind`'s definition below) and therefore exploit Howe's bisimulation even further. Our experiments with them have proven to be very fruitful.

Let us now define a few useful abstractions: let $\bot$ (bottom) be $\texttt{fix}(\lambda x.x)$, let $\pi_1(t)$ be $(\texttt{let}\ x, y = t\ \texttt{in}\ x)$, and let $\pi_2(t)$ be $(\texttt{let}\ x, y = t\ \texttt{in}\ y)$.

Free and bound variables are defined as usual. We write $t[x \backslash u]$ (and more generally $t[x_1 \backslash u_1; \cdots; x_n \backslash u_n]$) for the term t in which all the free occurrences of x have been replaced by u. Terms are identified up to alpha-equivalence.

Let `Top` be the following type: for all closed terms t_1 and t_2, $t_1 =_{\texttt{Top}} t_2$. `Top`'s equality is trivial because it identifies all elements. This type is especially useful to assign types to terms in contexts where their structure or behavior is irrelevant. When discussing types it is important to remember that a type is an equivalence relation on a set of terms and not simply a set of terms. Type A is a subtype of type B (written $A \sqsubseteq B$) if $x =_A y$ implies $x =_B y$. This means not only that every term in A is also in B, but that equality in A refines equality in B. Hence, $T \sqsubseteq \texttt{Top}$ for every type T. Sec. 3.1 discusses the type `Base`, which

[2] The only other values currently in Nuprl are tokens, atoms, and types, but more values can be added because the system is open-ended.

Core calculus:

$(\lambda x.F)\ a$	$\rightarrow$	$F[x\backslash a]$
$\texttt{let}\ x, y = \langle t_1, t_2\rangle\ \texttt{in}\ F$	$\rightarrow$	$F[x\backslash t_1; y\backslash t_2]$
$\texttt{case inl}(t)\ \texttt{of inl}(x) \Rightarrow F \mid \texttt{inr}(y) \Rightarrow G$	$\rightarrow$	$F[x\backslash t]$
$\texttt{case inr}(t)\ \texttt{of inl}(x) \Rightarrow F \mid \texttt{inr}(y) \Rightarrow G$	$\rightarrow$	$G[y\backslash t]$
$\texttt{if}\ \underline{n}_1 = \underline{n}_2\ \texttt{then}\ t_1\ \texttt{else}\ t_2$	$\rightarrow$	t_1, if $\underline{n}_1 = \underline{n}_2$
$\texttt{if}\ \underline{n}_1 = \underline{n}_2\ \texttt{then}\ t_1\ \texttt{else}\ t_2$	$\rightarrow$	t_2, if $\underline{n}_1 \neq \underline{n}_2$
$\texttt{fix}(t)$	$\rightarrow$	$t\ \texttt{fix}(t)$
$\texttt{let}\ x := t_1\ \texttt{in}\ t_2$	$\rightarrow$	$t_2[x\backslash t_1]$, if t_1 is a value

Canonical form tests:

$\texttt{ispair}(\langle t, t'\rangle, t_1, t_2)$	$\rightarrow t_1$	$\texttt{ispair}(v, t_1, t_2)$	$\rightarrow t_2$, if v is not a pair
$\texttt{isaxiom}(\texttt{Ax}, t_1, t_2)$	$\rightarrow t_1$	$\texttt{isaxiom}(v, t_1, t_2)$	$\rightarrow t_2$, if v is not axiom
$\texttt{islambda}(\lambda x.t, t_1, t_2)$	$\rightarrow t_1$	$\texttt{islambda}(v, t_1, t_2)$	$\rightarrow t_2$, if v is not a lambda
$\texttt{isinl}(\texttt{inl}(t), t_1, t_2)$	$\rightarrow t_1$	$\texttt{isinl}(v, t_1, t_2)$	$\rightarrow t_2$, if v is not a left injection
$\texttt{isinr}(\texttt{inr}(t), t_1, t_2)$	$\rightarrow t_1$	$\texttt{isinr}(v, t_1, t_2)$	$\rightarrow t_2$, if v is not a right injection
$\texttt{isint}(\underline{n}, t_1, t_2)$	$\rightarrow t_1$	$\texttt{isint}(v, t_1, t_2)$	$\rightarrow t_2$, if v is not an integer

Fig. 2: Nuprl's operational semantics

contains all Nuprl terms, but does not have this property (i.e. not every type T is a subtype of Base[3]) because equality on Base is Howe's bisimulation relation.

2.2 Operational Semantics

Fig. 2 presents some of Nuprl's reduction rules. This figure does not show the reduction rule for the call-by-valueall operator because it is slightly more complicated. This operator is like call-by-value but continues recursively evaluating subterms of pairs and injections.[4]

At any point in a computation, either a value is produced, or the computation is stuck, or we can take another step. For example, ($\texttt{let}\ x, y = \texttt{Ax}\ \texttt{in}\ F$) is a meaningless term that cannot evaluate further. It is stuck on the wrong kind of principal argument: Ax instead of a pair. Using the proof techniques presented below, in Sec. 4.3 we prove that this term is computationally equivalent to $\bot$. We can prove such results using ispair and isaxiom, and do not know of any other way discussed in the literature to accomplish this. Intuitively, we prove this lemma using the fact that isaxiom can compute to different values depending on whether its first argument computes to Ax or not. For example, $\texttt{isaxiom}(t, 0, 1)$ reduces to 0 if t is Ax and to 1 if t is, e.g., a pair. Note that even though they are computationally equal, ($\texttt{let}\ x, y = \texttt{Ax}\ \texttt{in}\ F$) and $\bot$ are fundamentally different in the sense that one could potentially detect whether a term is stuck (by slightly modifying our destructors such as *spread* or *decide*), but one cannot detect whether a term diverges or not.

[3] Being extentional, function types are in general not subtypes of Base.

[4] The call-by-valueall operator is similar to a restricted form of Haskell's *deepseq* operator. It can be defined using the other primitive operators (see the expanded version of this article at http://www.nuprl.org/Publications/), but for simplicity reasons we introduce it as a primitive in this paper.

2.3 Datatypes

Booleans As usual, we define Booleans using the disjoint union type as follows: $\mathbb{B} = \texttt{Unit} + \texttt{Unit}$. The `Unit` type is defined as $0 =_{\mathbb{Z}} 0$ and therefore `Ax` is its only inhabitant (up to computation). We define the Boolean true `tt` as `inl(Ax)`, and the Boolean false `ff` as `inr(Ax)`. Using the *decide* operator we define a conditional operator as follows: $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 = \texttt{case } t_1 \texttt{ of inl}(x) \Rightarrow t_2 \mid \texttt{inr}(x) \Rightarrow t_3$.

Lists We define lists as follows using Nuprl's union type [26] and recursive type [32] that allows one to build inductive types:[5] $\texttt{List}(T) = \texttt{rec}(L.\texttt{Unit} \cup T \times L)$. The type constructor $\cup$ creates the union of two types, not the disjoint union. The members of $A \cup B$ are members of A or B, not injections of them. A list is either a member of `Unit`, i.e., `Ax`, or a pair. The empty list `nil` is defined as `Ax`, and the cons operation, denoted by $\bullet$, as the pair constructor. We can distinguish an empty list and a non empty list because `Unit` and the product type are disjoint. Using `fix`, we define the following "list induction" operator:

$$\texttt{list\_ind}(L, b, f) =$$
$$\texttt{fix}(\lambda F.\lambda L.\texttt{ispair}(L, \texttt{let } h, t = L \texttt{ in } f\ h\ t\ (F\ t), \texttt{isaxiom}(L, b, \bot)))\ L$$

To define such a function that takes a list as input, we need to be able to test whether it is a pair or `Ax`. If we were to use the *spread* operator, we could destruct pairs, but computations would get stuck on `Ax` which we use to represent the empty list. Therefore, we need an operator such as the `ispair` canonical form test which allows us to perform two different computations depending on whether its first argument computes to a pair or not. Note that if `list_ind`'s first argument does not compute to a pair or to `Ax`, then the term diverges as opposed to returning an arbitrary value. This is necessary to prove untyped equivalences between list operations. We define the append and map operations as follows:

$$\begin{aligned} t_1 \texttt{ @ } t_2 &= \texttt{list\_ind}(t_1, t_2, \lambda h.\lambda t.\lambda r.h \bullet r) \\ \texttt{map}(f, t) &= \texttt{list\_ind}(t, \texttt{nil}, \lambda h.\lambda t.\lambda r.(f\ h) \bullet r) \end{aligned}$$

3 Computational Equivalence

3.1 Simulations and Bisimulations

Howe [23,24] defined the simulation or approximation relation $\leq$ using the following co-inductive rule: $t_1 \leq t_2$ if and only if (if t_1 computes to a canonical form $\Theta(u_1, \ldots, u_n)$ of the language defined in Sec. 2.1, then t_2 computes to a canonical form $\Theta(u'_1, \ldots, u'_n)$ such that for all $i \in \{1, \ldots, n\}$, $u_i \leq u'_i$). We say that t_1 approximates t_2 or that t_2 simulates t_1. This relation is reflexive (w.r.t. the terms defined in Sec. 2.1) and transitive. Howe then defined the bisimulation relation $\sim$ as the symmetric closure of $\leq$ (i.e., $t_1 \sim t_2$ iff $t_1 \leq t_2$ and $t_2 \leq t_1$), and proved that $\leq$ and $\sim$ are congruences w.r.t. Nuprl's computation system.[6]

[5] This new definition of lists replaces the one from Nuprl's book [16] where lists are considered as primitive objects. Using Nuprl's replay functionality, we were able to successfully replay the entire Nuprl library using this new definition of lists.

[6] Howe proved that $\sim$ is a congruence w.r.t. a lazy computation system by proving that all the operators of the system satisfy a property called *extensionality*. The expanded version of this article proves that the new operators introduced in this paper satisfy that property.

The following *context property* follows from the fact that $\sim$ is a congruence:

$$\forall i : \{1, \ldots, n\}.\ t_i \leq u_i \ \Rightarrow\ G[x_1 \backslash t_1; \cdots ; x_n \backslash t_n] \leq G[x_1 \backslash u_1; \cdots ; x_n \backslash u_n]$$

Howe's bisimulation relation respects computation, i.e., if $t_1 \sim t_2$ then (1) t_1 computes to a value iff t_2 computes to a value, and (2) if t_1 computes to a value v_1 then t_2 computes to a value v_2 with same outer operator such that $v_1 \sim v_2$.

Because $\bot$ does not compute to a canonical form, by definition $\bot \leq t$ is true for any term t; hence for example $\langle \texttt{Ax}, \bot \rangle \leq \langle \texttt{Ax}, \texttt{Ax} \rangle$. Similarly, because `Ax` is not a pair, (`let` $x, y =$ `Ax in` x) does not compute to a canonical form, and by definition, `let` $x, y =$ `Ax in` $x \leq t$ is true for any term t (we prove `let` $x, y =$ `Ax in` $F \sim \bot$ in Sec. 4.3). However, `Ax` $\leq \bot$ is not true because $\bot$ diverges while `Ax` is a value; hence $\langle \texttt{Ax}, \texttt{Ax} \rangle \leq \langle \texttt{Ax}, \bot \rangle$ is not true either.

Let us write `halts`(t) if t reduces to a value—we say that t converges. We can define convergence using call-by-value because the call-by-value operator (`let` $x :=$ t_1 `in` t_2) first evaluates t_1. The term t_1 converges if and only if the term (`let` $x :=$ t_1 `in Ax`) evaluates to `Ax`. So we simply define `halts`(t) to be the simulation `Ax` $\leq$ (`let` $x :=$ t `in Ax`). Because `Ax` is a canonical value then `Ax` $\leq$ (`let` $x :=$ t `in Ax`) is true if and only if (`let` $x :=$ t `in Ax`) computes to `Ax`, i.e., if and only if t computes to a value.

Constable and Smith [35,17] introduced partial types to reason about computations that might not halt. For any type T, the partial type $\overline{T}$ contains all members of T as well as all divergent terms, and has the following equality: two terms are equal in $\overline{T}$ if they have the same convergence behavior (i.e., either neither computes to a value or both compute to a value), and when they converge, they are equal in T. An important partial type is `Base` $= \overline{\texttt{Value}}$ where `Value` is the type of all closed canonical terms of the computation system with $\sim$ as its equality. Because `Base` is a partial type, it contains converging as well as diverging terms, and equal terms have the same convergence behavior.

3.2 Simple Facts About Lists

Sec. 4 proves that for all terms f and t in `Top`, `map`(f, t) `@ nil` $\sim$ `map`(f, t).

If t is a list, the first expression (`map`(f, t) `@ nil`) requires two passes over the list t while the second expression (`map`(f, t)) requires only one. This simple bisimulation will be our running example to illustrate the techniques we use to optimize our distributed processes (discussed in Sec. 5).

Note that this lemma would be easy to prove by induction on the list t if we were using the list type instead of `Top`. However, we might need to instantiate t with a term for which it would be non-trivial to prove that it is a list because Nuprl is based on an extension of the *untyped* λ-calculus and type inference and type checking are *undecidable*. In addition, if we were to use a typed equality (instead of $\sim$) for substitution in some context, then we would also have to prove that the context is functional over the type of the equality. That is, to rewrite in the term $C[t]$ of type B using $t =_A u$, we have to prove that $\lambda z.C[z]$ is of type $A \rightarrow B$. Moreover, the above equivalence is indeed true for any term t, e.g., it is true when t is a stream.

Note that it is not true that for all terms t, t `@` `nil` $\sim t$. For example, by definition of `@`, $(\lambda x.x)$ `@` `nil` $\sim \bot$. However, the bisimulation $\lambda x.x \sim \bot$ is not true because the simulation $\lambda x.x \leq \bot$ is not true. This shows that there are some terms t for which $t \leq t$ `@` `nil` does not hold.[7] However, Sec. 4 proves that for all terms t, t `@` `nil` $\leq t$. A corollary of that lemma is that `map`(f, t) `@` `nil` $\leq$ `map`(f, t).

4 Proof Techniques

This section presents three proof techniques we use to prove bisimulations: Crary's least upper bound property [18], patterns of reasoning regarding our new canonical form tests, and patterns of reasoning regarding our `halts` operator. It also presents three derived proof techniques called lifting, normalization, and strictness. Using these techniques, we prove `map`(f, t) `@` `nil` $\sim$ `map`(f, t), and in Sec. 5, we optimize distributed processes.

4.1 Least Upper Bound Property

Using the properties of $\leq$ and that `fix`$(f) = f$ `fix`(f), it is easy to prove by induction on n that $\forall n : \mathbb{N}.\ f^n(\bot) \leq$ `fix`(f) [18]. So `fix`(f) is an upper bound of its approximations. The least upper bound property [18, Theorem 5.9] is:

Rule `[least-upper-bound].` $\forall n : \mathbb{N}.\ G(f^n(\bot)) \leq t \ \Rightarrow\ G($`fix`$(f)) \leq t$.

4.2 Canonical Form Tests

In order to reason about its programs, we gave Nuprl the ability to reason about the *canonical form tests* such as `ispair`, `isaxiom`, etc.[8] These effective operations on `Base` allow us to reason in the programming language, where in the past we resorted to reflection in the logic [6].

Membership Rules

Rule `[ispair-member].` *To prove that* `ispair`$(t_1, t_2, t_3) \in T$*, it is enough to prove* `halts`(t_1)*, and that both* t_2 *and* t_3 *are members of* T*.*

We introduce similar rules for the other canonical form tests. Using this rule we can trivially prove the following fact:

Lemma 1. *For all terms* t *in* `Base`*, if* `halts`(t) *then* `ispair`$(t,$ `tt`$,$ `ff`$) \in \mathbb{B}$.

The same is true for the other tests. Using these facts, we can, e.g., decide whether a converging term is a pair or not.

Semi-decision Rules Depending on how `ispair` computes we can deduce various pieces of information. If we know that `ispair`(t_1, t_2, t_3) always computes to t_2 and cannot compute to t_3 then we know that t_1 is a pair. If we know that `ispair`(t_1, t_2, t_3) always computes to t_3 and cannot compute to t_2 then we know that t_1 is not a pair. These properties are captured by the following two rules:

[7] The expanded version of this article provides a characterization of the terms that satisfy that property.

[8] The proofs that the rules introduced in this section are valid w.r.t. Allen's PER (Partial Equivalence Relations) semantics [2,3] are presented in the expanded version of this article.

```
⊢ ∀F:Top. (let x,y = Ax in F[x;y]) ~ bottom())
|
BY (SqReasoning
    THEN Assert ⌜(if Ax is a pair then 0 otherwise 1) = 1⌝⋅
    THEN (Reduce 0 THEN Auto THEN AutoPairEta [2;1] (-1)))
```

Fig. 3: Computational equivalence between $\bot$ and a stuck term

Rule `[ispair]`. *To prove* $t \in \texttt{Top} \times \texttt{Top}$ *(i.e., t is a pair), it is enough to prove* $\texttt{ispair}(t, \texttt{inl}(a), \texttt{inr}(b)) \sim \texttt{inl}(a)$ *for some terms a and b.*

Rule `[not-ispair]`. *To prove* $\texttt{ispair}(t_1, t_2, t_3) \sim t_3$*, it is enough to prove that* $\texttt{ispair}(t_1, \texttt{inl}(a), \texttt{inr}(b)) \sim \texttt{inr}(b)$ *for some terms a and b.*

We introduce similar rules for the other canonical form tests. Using these rules we can prove such results as (similar results are true for the other tests):

Lemma 2. *For all terms t, a, b in* `Base`*, if* $\texttt{halts}(t)$ *then* $t \sim \langle \pi_1(t), \pi_2(t) \rangle \ \vee \ \texttt{ispair}(t, a, b) \sim b$.

Proof. By Lemma 1, $\texttt{ispair}(t, \texttt{tt}, \texttt{ff}) \in \mathbb{B}$. Therefore, either $\texttt{ispair}(t, \texttt{tt}, \texttt{ff}) \sim \texttt{tt}$ or $\texttt{ispair}(t, \texttt{tt}, \texttt{ff}) \sim \texttt{ff}$ (this is true for any Boolean). If $\texttt{ispair}(t, \texttt{tt}, \texttt{ff}) \sim \texttt{tt}$ then using rule `[ispair]` we obtain that t is a pair and therefore $t \sim \langle \pi_1(t), \pi_2(t) \rangle$. If $\texttt{ispair}(t, \texttt{tt}, \texttt{ff}) \sim \texttt{ff}$ then using rule `[not-ispair]` we obtain that $\texttt{ispair}(t, a, b) \sim b$. □

4.3 Convergence

Rule `[convergence]`. *To prove $t_1 \leq t_2$, one can assume* $\texttt{halts}(t_1)$.

This rule follows directly from $\leq$'s definition. For example, to prove $\texttt{let } x, y = p \texttt{ in } F \leq \texttt{let } x, y = q \texttt{ in } G$, one can assume that $\texttt{halts}(\texttt{let } x, y = p \texttt{ in } F)$.

Nuprl also has rules to reason about $\texttt{halts}(t)$. If a non-canonical term converges, then its principal arguments have to converge to the appropriate canonical forms as presented in Fig 2. For example the following two rules follow from the operational semantics of *spread* and `ispair` (we have similar rules for the other non-canonical operators):

Rule `[halt-spread]`. *If* $\texttt{halts}(\texttt{let } x, y = p \texttt{ in } F)$ *then p computes to a pair.*

Rule `[halt-ispair]`. *If* $\texttt{halts}(\texttt{ispair}(t_1, t_2, t_3))$ *then* $\texttt{halts}(t_1)$.

Let us go back to the example presented in Sec. 2.2. We now have enough tools to prove the following lemma:

Lemma 3. *For all terms F in* `Top`*,* $\texttt{let } x, y = \texttt{Ax in } F \sim \bot$

Proof. Fig 3 presents our Nuprl proof of that fact. That proof goes as follows: By definition of $\sim$, we have to prove $\texttt{let } x, y = \texttt{Ax in } F \leq \bot$ and $\bot \leq \texttt{let } x, y = \texttt{Ax in } F$. The second simulation is trivial. Let us prove the first one. Using `[convergence]`, we can assume $\texttt{halts}(\texttt{let } x, y = \texttt{Ax in } F)$ and using `[halt-spread]`, that `Ax` is a pair. This reasoning is done by our `SqReasoning` tactic. Finally, the term $\texttt{ispair}(\texttt{Ax}, 0, 1)$ computes to 1, and because we deduced that `Ax` is a pair, it also reduces to 0, and we have an absurdity. □

4.4 Lifting

Now we describe the following derived proof techniques: lifting, normalization (see Sec. 4.5 below), and strictness (see Sec. 4.6 below) which are used in Sec. 4.7 below. Lifting transforms a term t into t' such that $t \sim t'$ and such that t' has a smaller path to the principal argument of a subterm of t. Let us now provide a few examples. The following bisimulation specifies a lifting operation, where the path to p is shorter in the second term than in the first term:

Lemma 4. *For all terms F and G in* `Top`*:*

$$\texttt{let } c, d = (\texttt{let } a, b = p \texttt{ in } F) \texttt{ in } G \sim \texttt{let } a, b = p \texttt{ in } (\texttt{let } c, d = F \texttt{ in } G)$$

Proof. To prove that bisimulation, we have to prove that the first term simulates the second one and vice versa. Let us prove that the second one simulates the first one (the other direction is similar), i.e., $\texttt{let } c, d = (\texttt{let } a, b = p \texttt{ in } F) \texttt{ in } G \leq \texttt{let } a, b = p \texttt{ in } (\texttt{let } c, d = F \texttt{ in } G)$. Using `[convergence]`, we can assume $\texttt{halts}(\texttt{let } c, d = (\texttt{let } a, b = p \texttt{ in } F) \texttt{ in } G)$, from which, using `[halt-spread]` twice, we obtain that p is a pair. More precisely, we can prove that p is the pair $\langle \pi_1(p), \pi_2(p) \rangle$. By replacing p by $\langle \pi_1(p), \pi_2(p) \rangle$ in the above simulation, and by reducing both sides, we obtain $\texttt{let } c, d = F[a \backslash \pi_1(p); b \backslash \pi_2(p)] \texttt{ in } G \leq \texttt{let } c, d = F[a \backslash \pi_1(p); b \backslash \pi_2(p)] \texttt{ in } G$, which is true by reflexivity of $\leq$. □

Using this lemma, one can, e.g., derive the following chain of rewrites:

$$\begin{aligned} &\texttt{let } a, b = (\texttt{let } c, d = p \texttt{ in } \langle c, d \rangle) \texttt{ in } F \\ &\sim \texttt{let } c, d = p \texttt{ in } (\texttt{let } a, b = \langle c, d \rangle \texttt{ in } F) \\ &\quad \sim \texttt{let } c, d = p \texttt{ in } F[a \backslash c; b \backslash d] \end{aligned}$$

The following bisimulation specifies another lifting operation where the path to t_1 is shorter in the second term than in the first one:

Lemma 5. *For all terms t_1, t_2, t_3, t_4, and t_5 in* `Top`*:*

$$\begin{aligned} &\texttt{ispair}(\texttt{ispair}(t_1, t_2, t_3), t_4, t_5) \\ &\sim \texttt{ispair}(t_1, \texttt{ispair}(t_2, t_4, t_5), \texttt{ispair}(t_3, t_4, t_5)) \end{aligned}$$

The proof of this is similar to the proof of Lemma 4. Because lifting does not always result in a smaller term it must therefore be used in a controlled way.

4.5 Normalization

Normalization allows one to make use of the information given by destructors such as spread or decide, i.e., that some terms are forced to be pairs or injections by the computation system. Normalization achieves some kind of common subexpression elimination, which is a standard optimization technique. For example, the next lemma says that the expression on left-hand-side has a value if and only if p (which can be an arbitrary large term) is a pair, and more precisely in F it has to be the pair $\langle a, b \rangle$:

Lemma 6. *For all terms p and F in* `Top`*:*

$$\texttt{let } x, y = p \texttt{ in } F[z \backslash p] \sim \texttt{let } x, y = p \texttt{ in } F[z \backslash \langle x, y \rangle]$$

The proof of this is similar to the proof of Lemma 4.

4.6 Strictness

Strictness says that if $\bot$ is one of the principal arguments of a term then this term is computationally equal to $\bot$. For example we proved the following lemma:

Lemma 7. *For all terms F in* `Top`*,* $(\texttt{let}\ x, y = \bot\ \texttt{in}\ F) \sim \bot$.

The proof of this is similar to the proof of Lemma 4. Intuitively, such lemmas are true because to evaluate a non-canonical term, one has to evaluate its principal arguments. If one of these principal arguments is $\bot$, then the computation diverges. Therefore, the entire term is computationally equal to $\bot$.

4.7 Back To Our List Example

As explained in Sec. 3.2, to prove $\texttt{map}(f, t)\ \texttt{@}\ \texttt{nil} \sim \texttt{map}(f, t)$, we first prove the following lemma:

Lemma 8. *For all terms t in* `Top`*,* $t\ \texttt{@}\ \texttt{nil} \leq t$.

Proof. Because `@` is defined using `fix` (see Sec. 2.3), we prove that lemma using the `[least-upper-bound]` rule (see Sec.4.1). We now have to prove that any approximation of the fixpoint is simulated by t. Let

$$F = \lambda F.\lambda L.\texttt{ispair}(L, \texttt{let}\ x, y = L\ \texttt{in}\ x \bullet (F\ y), \texttt{isaxiom}(L, \texttt{nil}, \bot))$$

We have $(t\ \texttt{@}\ \texttt{nil}) = (\texttt{fix}(F)\ t)$ by definition of append and beta-reduction. We have to prove that for all natural numbers n, and for all terms t,

$$F^n\ \bot\ t \leq t$$

which we prove by induction on n. The base case boils down to proving that $\bot\ t \leq t$ which is true using strictness. In the interesting induction case, assuming that for all terms t, $F^{n-1}\ \bot\ t \leq t$, we have to prove $F\ (F^{n-1}\ \bot)\ t \leq t$, i.e.,

$$\texttt{ispair}(t, \texttt{let}\ x, y = t\ \texttt{in}\ x \bullet ((F^{n-1}\ \bot)\ y), \texttt{isaxiom}(t, \texttt{nil}, \bot)) \leq t \quad (1)$$

Let P be $\texttt{ispair}(t, \texttt{let}\ x, y = t\ \texttt{in}\ x \bullet ((F^{n-1}\ \bot)\ y), \texttt{isaxiom}(t, \texttt{nil}, \bot))$. Using `[convergence]`, we can assume $\texttt{halts}(P)$. Using `[halt-ispair]`, we obtain $\texttt{halts}(t)$. By Lemma 2, we get $t \sim \langle \pi_1(t), \pi_2(t) \rangle$ or $P \sim \texttt{isaxiom}(t, \texttt{nil}, \bot)$.

If $t \sim \langle \pi_1(t), \pi_2(t) \rangle$, we have to prove the following simulation obtained from simulation 1 by replacing t by $\langle \pi_1(t), \pi_2(t) \rangle$ and by reducing:

$$\pi_1(t) \bullet ((F^{n-1}\ \bot)\ \pi_2(t)) \leq \langle \pi_1(t), \pi_2(t) \rangle$$

Because the cons operator is defined as the pair constructor, by the context property it remains to prove the following simulation, which is true by induction hypothesis: $((F^{n-1}\ \bot)\ \pi_2(t)) \leq \pi_2(t)$.

If $P \sim \texttt{isaxiom}(t, \texttt{nil}, \bot)$, we have to prove $\texttt{isaxiom}(t, \texttt{nil}, \bot) \leq t$. Using the version of Lemma 2 for `isaxiom`, we obtain $t \sim \texttt{Ax}$ or $\texttt{isaxiom}(t, \texttt{nil}, \bot) \sim \bot$. Both cases are trivial: in the first case we have to prove $\texttt{Ax} \leq \texttt{Ax}$ and in the second we have to prove $\bot \leq t$. □

Let us now prove the lemma we set out to prove in Sec. 3.2:

Lemma 9. *For all terms t and f in* Top, $\texttt{map}(f,t)\ \texttt{@}\ \texttt{nil} \sim \texttt{map}(f,t)$.

Proof. By definition of $\sim$, we have to prove $\texttt{map}(f,t)\ \texttt{@}\ \texttt{nil} \leq \texttt{map}(f,t)$ (which is true by Lemma 8), and $\texttt{map}(f,t) \leq \texttt{map}(f,t)\ \texttt{@}\ \texttt{nil}$. Because map is a fixpoint, we can prove the latter using the [least-upper-bound] rule. Let

$$F = \lambda F.\lambda L.\texttt{ispair}(L, \texttt{let}\ x,y = L\ \texttt{in}\ (f\ x) \bullet (F\ y), \texttt{isaxiom}(L,\texttt{nil},\bot))$$

We then have to prove that for all natural numbers n and for all terms f and t,

$$F^n\ \bot\ t \leq \texttt{map}(f,t)\ \texttt{@}\ \texttt{nil}$$

which we prove by induction on n. Once again, the base case is trivial. Assume that for all terms t, $F^{n-1}\ \bot\ t \leq \texttt{map}(f,t)\ \texttt{@}\ \texttt{nil}$, we have to prove that $F\ (F^{n-1}\ \bot)\ t \leq \texttt{map}(f,t)\ \texttt{@}\ \texttt{nil}$, i.e., we have to prove the following simulation:

$$\begin{array}{l}\texttt{ispair}(t, \texttt{let}\ x,y = t\ \texttt{in}\ (f\ x) \bullet ((F^{n-1}\ \bot)\ y), \texttt{isaxiom}(t,\texttt{nil},\bot)) \\ \leq \texttt{map}(f,t)\ \texttt{@}\ \texttt{nil}\end{array} \quad (2)$$

Let $P = \texttt{ispair}(t, \texttt{let}\ x,y = t\ \texttt{in}\ (f\ x) \bullet \texttt{map}(f,y), \texttt{isaxiom}(t,\texttt{nil},\bot))$, which is $\texttt{map}(f,t)$ unfolded once. We obtain the following sequence of bisimulations by unfolding the definitions of map and @ in $(\texttt{map}(f,t)\ \texttt{@}\ \texttt{nil})$:

$$\begin{array}{l}\texttt{map}(f,t)\ \texttt{@}\ \texttt{nil} \sim P\ \texttt{@}\ \texttt{nil} \\ \sim \texttt{ispair}(P, \texttt{let}\ x,y = t\ \texttt{in}\ x \bullet (y\ \texttt{@}\ \texttt{nil}), \texttt{isaxiom}(P,\texttt{nil},\bot))\end{array}$$

Using lifting (Lemma 5) and normalization, we obtain the following bisimulation:

$$\begin{array}{l}\texttt{map}(f,t)\ \texttt{@}\ \texttt{nil} \\ \sim \texttt{ispair}(t, \texttt{let}\ x,y = t\ \texttt{in}\ (f\ x) \bullet (\texttt{map}(f,y)\ \texttt{@}\ \texttt{nil}), \texttt{isaxiom}(t,\texttt{nil},\bot))\end{array}$$

Therefore, given that we have to prove simulation 2, it means that we have to prove the following simulation:

$$\begin{array}{l}\texttt{ispair}(t, \texttt{let}\ x,y = t\ \texttt{in}\ (f\ x) \bullet ((F^{n-1}\ \bot)\ y), \texttt{isaxiom}(t,\texttt{nil},\bot)) \\ \leq \texttt{ispair}(t, \texttt{let}\ x,y = t\ \texttt{in}\ (f\ x) \bullet (\texttt{map}(f,y)\ \texttt{@}\ \texttt{nil}), \texttt{isaxiom}(t,\texttt{nil},\bot))\end{array}$$

which is true by induction hypothesis and the context property. □

5 Process Optimization

Nuprl implements a Logic of Events (LoE) [10,12,13] to specify and reason about distributed programs, as well as a General Process Model (GPM) [11] to implement them. We have proved a direct relationship between some LoE combinators and some GPM combinators. This allows us to automatically generate processes that are guaranteed to satisfy the logical specifications of LoE.

Using the proof techniques presented in the above section, we were able to optimize many automatically generated GPM processes. For example, we optimized our synthesized version of Paxos, which is used by the ShadowDB replicated database [34]. Because our synthesized Paxos was initially too slow, it was

only used to handle database failures, which are critical to handle correctly but are not frequent. When a failure occurs, Paxos ensures that the replicas agree on the next set of replicas. We can now also use Paxos to consistently order the transactions of the replicated databases. Initially, our synthesized code could only handle one transaction every few seconds. Thanks to our automatic optimizer, the code we synthesize is now about an order of magnitude faster. Our goal is to be able to handle several thousands of transactions per second. Even though we have not yet reached that goal, this work is already an encouraging first step towards generating fast correct-by-construction code.

A GPM process is modeled as a function that takes inputs and computes a new process as well as outputs. For distributed programs based on message passing, these inputs and outputs are messages. Formally, a process that takes inputs of type A, and outputs elements of type B, is an element of (a variant of) the following co-recursive type:

$$\texttt{corec}(\lambda P.A \to P \times \texttt{Bag}(B))$$

where `corec` is defined as follows:

$$\texttt{corec}(G) = \cap n : \mathbb{N}.\texttt{fix}(\lambda P.\lambda n.\texttt{if } n = 0 \texttt{ then Top else } G\ (P\ (n-1)))\ n$$

Note the use of bags, also called multisets, formally defined as quotiented lists. The reason for using that type is outside the scope of this paper. However, let us mention that processes can output more than one element and these elements need not be ordered. In the rest of this paper, we use curly braces to denote specific bag instances. Lists and bags have many similar operations such as: `bmap` the map operation on bags, `bnull` the null operation, `bconcat` the concat operation which flattens bags of bags, and `>>=` the bind operation of the bag monad, defined as b `>>=` f = $\texttt{bconcat}(\texttt{bmap}(f, b))$. For example, $(\{1;2;2;4\}$ `>>=` $\lambda x.\{x; x+1\}) = \{1;2;2;3;2;3;4;5\} = \{1;2;2;2;3;3;4;5\}$.

Many of the GPM combinators are defined using `fix`. Because processes are typically defined using several combinators, fixpoints end up being deeply nested which affects the computational complexity of the processes. Using, among other things, the least upper bound property, we can often reduce the number of fixpoints occurring in processes. This is our main process optimization technique.

Let us now present some GPM combinators. Processes often need to maintain an internal state. Therefore, the combinators defined below will all be of the form $\texttt{fix}(\lambda F.\lambda s.\lambda m.G)\ init$, where $init$ is an initial state, and G is a transition function that takes the current state of the process (s) and an input (m), and generates a new process and some output.

5.1 Combinators

Base Combinator It builds a process that applies a function to its inputs:

$$\texttt{base}(f) = \texttt{fix}(\lambda F.\lambda s.\lambda m.\langle F\ s, f\ m\rangle)\ \texttt{Ax}$$

Base processes are stateless, which is modeled using the term `Ax` as the state of the base combinator.

Composition Combinator It builds a process that applies a function f to the outputs of its sub-component X:

$$f \texttt{ o } X = \texttt{fix}(\lambda F.\lambda X.\lambda m. \begin{pmatrix} \texttt{let } X', out = X\ m \texttt{ in} \\ \texttt{let } out' ::= \texttt{bmap}(f, out) \texttt{ in} \\ \langle F\ X', out' \rangle \end{pmatrix})\ X$$

The state maintained by f o X is the state maintained by X. Note that for efficiency issues, we use the call-by-valueall operator ::= in order to generate the outputs out'.

Buffer Combinator From an initial buffer *init* and a process X producing transition functions, this combinator builds a process that buffers its outputs:

$$\begin{array}{l} \texttt{buffer}(X, init) = \\ \texttt{fix}(\lambda F.\lambda s.\lambda m. \begin{pmatrix} \texttt{let } X, buf = s \texttt{ in} \\ \texttt{let } X', b = X\ m \texttt{ in} \\ \texttt{let } b' ::= b \texttt{ >>= } \lambda f.(buf \texttt{ >>= } f) \texttt{ in} \\ \langle F\ \langle X', \texttt{if bnull}(b') \texttt{ then } buf \texttt{ else } b' \rangle, b' \rangle \end{pmatrix})\ \langle X, init \rangle \end{array}$$

The state maintained by $\texttt{buffer}(X, init)$ is the pair of the state maintained by X and its previous outputs (initially *init*).

5.2 Example

The following process uses the three combinators presented above:

$$P = \texttt{buffer}((\lambda n.\lambda buf.\{n + buf\}) \texttt{ o } \texttt{base}(\lambda m.\{m\}), \{0\})$$

This process maintains a state constituted of a single integer, initialized to 0. Its inputs are integers. At any point in time, its state is the sum of all the inputs it has received in the past. Because the combinators used in P are defined as fixpoints, P contains three nested occurrences of `fix`. We will now show that P is computationally equivalent to the following even simpler process:

$$P' = \texttt{fix}(\lambda F.\lambda s.\lambda m.\texttt{let } x ::= m + s \texttt{ in } \langle F\ x, \{x\} \rangle)\ 0$$

Using Nuprl's powerful tactic mechanism we automatically generate P' from P, and we automatically prove that $P \sim P'$. Our experiments showed that it takes between 100 and 200 computation steps for P to process a single input while it takes less than 10 computation steps for P' to process a single input.

Standard Form To optimize our processes we take advantage of the fact that many of them are of the following form:

$$\begin{array}{l} \texttt{process}(n, L, S, R, I) = \\ \texttt{fix}(\lambda F.\lambda s.\lambda m. \begin{pmatrix} \texttt{let } x_1 ::= L\ s\ m\ 1 \texttt{ in} \\ \cdots \\ \texttt{let } x_n ::= L\ s\ m\ n\ x_1\ \cdots\ x_{n-1} \texttt{ in} \\ \langle F\ (S\ s\ m\ x_1\ \cdots\ x_n), R\ s\ m\ x_1\ \cdots\ x_n \rangle \end{pmatrix})\ I \end{array}$$

where L is a sequence of instructions defined as a function, n is the number of instructions that the process executes on each input, S computes the next state of the process, R computes the outputs, and I is its initial state.

Transformations We prove the next three lemmas using the same proof technique as in Sec. 4.7. These lemmas show that if processes are built using the base, composition, and buffer combinators (and many other primitive combinators of the GPM not presented in this paper), then they are guaranteed to be of the standard form $\texttt{process}(n, L, S, R, I)$.

Lemma 10. *Given a term f of type* Top*, the following bisimulation is true:*

$$\texttt{base}(f) \sim \texttt{process}(0, \lambda x.\bot, \lambda s.\lambda m.\texttt{Ax}, \lambda s.\lambda m.f\ m, \texttt{Ax})$$

Lemma 11. *Given f, L, S, R, and I terms of type* Top*, and n a natural number, the following bisimulation is true:*

$$\begin{array}{l} f\ \texttt{o}\ \texttt{process}(n, L, S, R, I) \\ \sim \texttt{process}(\ n+1, \\ \qquad \lambda s.\lambda m.\lambda i.\texttt{if}\ i = n+1\ \texttt{then}\ \lambda x_1 \ldots . \lambda x_n.\ \texttt{bmap}(f, R\ s\ m\ x_1\ \cdots\ x_n)\,, \\ \qquad\qquad\qquad \texttt{else}\ L\ s\ m \\ \qquad \lambda s.\lambda m.\lambda x_1 \ldots . \lambda x_n.\ \lambda x_{n+1}.S\ s\ m\ x_1\ \cdots\ x_n, \\ \qquad \lambda s.\lambda m.\lambda x_1 \ldots . \lambda x_n.\ \lambda x_{n+1}.x_{n+1}, \\ \qquad I) \end{array}$$

Lemma 12. *Given L, S, R, I, and I' terms of type* Top*, and n a natural number, the following bisimulation is true:*

$$\begin{array}{l} \texttt{buffer}(\texttt{process}(n, L, S, R, I), I') \\ \sim \texttt{process}(\ n+1, \\ \qquad \lambda s.\lambda m.\lambda i.\texttt{if}\ i = n+1 \\ \qquad\qquad\qquad \texttt{then}\ \lambda x_1 \ldots . \lambda x_n.\ (R\ \pi_1(s)\ m\ x_1\ \cdots\ x_n)\ \texttt{>>=}\ \lambda f.(\pi_2(s)\ \texttt{>>=}\ f) \\ \qquad\qquad\qquad \texttt{else}\ L\ \pi_1(s)\ m \\ \qquad \lambda s.\lambda m.\lambda x_1 \ldots . \lambda x_n.\ \lambda x_{n+1}.\langle S\ \pi_1(s)\ m\ x_1\ \cdots\ x_n \\ \qquad\qquad\qquad\qquad\qquad , \texttt{if bnull}(x_{n+1})\ \texttt{then}\ s\ \texttt{else}\ x_{n+1}\rangle \\ \qquad \lambda s.\lambda m.\lambda x_1 \ldots . \lambda x_n.\ \lambda x_{n+1}.x_{n+1}, \\ \qquad \langle I, I'\rangle) \end{array}$$

Transformation of P into P' Using the bisimulations presented above, we automatically rewrite P into P', and because our bisimulations are untyped, proving that P is computationally equivalent to P' is also trivial: it only requires us to prove that some terms are in Top, and all closed terms are trivially in Top.

6 Related Work and Conclusion

This paper describes computational proof techniques based on bisimulations which we use in the Nuprl proof assistant in order to optimize distributed processes (programs in general). McCarthy [31] recognized the value of type free reasoning, and we took that to heart in the design of CTT by providing type free rules about computation, called "direct computation rules". Now we know that this kind of reasoning can be made even richer.

Gordon [20] characterizes contextual equivalence as some form of co-inductively defined bisimulation. Using co-inductive reasoning, Gordon can easily prove, e.g., various bisimulations between streams. For example, he proves that $\texttt{iterate}(f, f\ x)$

and $\texttt{map}(f, \texttt{iterate}(f, x))$ are bisimilar, where $\texttt{iterate}(f, x)$ is defined in Nuprl as $\texttt{fix}(\lambda F.\lambda x.\langle x, F\ (f\ x)\rangle)\ x$. Nuprl's corresponding method to prove such results is the least upper bound property. Gordon proves this result using a co-inductive reasoning, while we prove it by induction on the natural number we obtain by approximating the two fixpoints used to define `map` and `iterate`. Apart from that difference, the resulting proofs are similar in spirit.

Note that we have not yet formally proved that the processes returned by our optimizer have a better complexity than the processes it takes as inputs. Using Isabelle/HOL, Aspinall, Beringer, and Momigliano [5] developed an optimization validation technique, based on a proof-carrying code approach, to prove that optimized programs use less resources than the non-optimized versions. Currently, we cannot measure the complexity of programs inside Nuprl because if t_1 reduces to t_2 then $t_1 \sim t_2$, and hence we cannot distinguish between them in any context.

We hope to solve this issue by either using some kind of reflection, or introducing a subtype of `Base` where equality would be alpha-equality. Also, in order to enhance the usability of our processes in industrial strength systems, we need to identify and verify other optimizations. As mentioned in Sec. 1, we view this work as a step towards making Nuprl a usable programming framework. In the meantime, we have built a Lisp translator for our processes.

In the last two decades, much work has been done on compiler verification. See Dave [19] for earlier references. To name a few: Using Coq, Leroy has developed and certified a compiler for a C-like language [29]. He generated the compiler using Coq's extraction mechanism to Caml code. The compiler is certified thanks to "a machine-checked proof of semantic preservation" [29]. Also using Coq, Chlipala [15] developed a verified compiler for an impure functional programming language with references and exceptions that produces code in an idealized assembly language. He proved the correctness of the compiler using a big-step operational semantics. Li [30] designed a verified compiler in HOL, from an high-level ML-like programming language implemented in HOL to ARM assembly code. Each transformation of the compiler generates a correctness argument along with a piece of code.

Following this line of work, we now would like to tackle the task of building a verified compiler for Nuprl in Nuprl.

Acknowledgements

We would like to thank our colleagues Professor Robert L. Constable, David Guaspari, and Evan Moran for their helpful criticism.

References

1. The Coq Proof Assistant. `http://coq.inria.fr/`.
2. Stuart F. Allen. A non-type-theoretic definition of martin-löf's types. In *LICS*, pages 215–221. IEEE Computer Society, 1987.
3. Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.
4. Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. *J. Applied Logic*, 4(4):428–469, 2006.

5. David Aspinall, Lennart Beringer, and Alberto Momigliano. Optimisation validation. *Electr. Notes Theor. Comput. Sci.*, 176(3):37–59, 2007.
6. Eli Barzilay. *Implementing Reflection in* NUPRL. PhD thesis, Cornell University, 2006.
7. Stefan Berghofer. Program extraction in simply-typed higher order logic. In *Types for Proofs and Programs, 2nd Int'l Workshop, TYPES*, volume 2646 of *LNCS*, pages 21–38. Springer, 2002.
8. Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In *Types for Proofs and Programs, Int'l Workshop, TYPES*, volume 2277 of *LNCS*, pages 24–40. Springer, 2000.
9. Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.
10. Mark Bickford. Component specification using event classes. In *Component-Based Software Engineering, 12th Int'l Symp.*, volume 5582 of *LNCS*, pages 140–155. Springer, 2009.
11. Mark Bickford, Robert Constable, and David Guaspari. Generating event logics with higher-order processes as realizers. Technical report, Cornell University, 2010.
12. Mark Bickford and Robert L. Constable. Formal foundations of computer security. In *NATO Science for Peace and Security Series, D: Information and Communication Security*, volume 14, pages 29–52. 2008.
13. Mark Bickford, Robert L. Constable, and Vincent Rahli. Logic of events, a framework to reason about distributed systems. In *Languages for Distributed Algorithms Workshop*, 2012.
14. B. Charron-Bost and A. Schiper. The Heard-Of model: computing in distributed systems with benign failures. *Distributed Computing*, 22(1):49–71, 2009.
15. Adam Chlipala. A verified compiler for an impure functional language. In *37th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 93–106. ACM, 2010.
16. R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.
17. Robert L. Constable and Scott F. Smith. Computational foundations of basic recursive function theory. *Theoretical Computer Science*, 121(1&2):89–112, 1993.
18. Karl Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, Ithaca, NY, August 1998.
19. Maulik A. Dave. Compiler verification: a bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6):2, 2003.
20. Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Electr. Notes Theor. Comput. Sci.*, 1:232–252, 1995.
21. Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL - a modular logical environment. In *TPHOLs*, volume 2758 of *LNCS*, pages 287–303. Springer, 2003.
22. Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.
23. Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of Fourth IEEE Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society, 1989.
24. Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Inf. Comput.*, 124(2):103–112, 1996.
25. Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *LICS*, pages 86–95. IEEE Computer Society, 2003.
26. Alexei Kopylov. *Type Theoretical Foundations for Data Structures, Classes, and Objects*. PhD thesis, Cornell University, Ithaca, NY, 2004.
27. Christoph Kreitz. *The Nuprl Proof Development System, Version 5, Reference Manual and User's Guide*. Cornell University, Ithaca, NY, 2002. `www.nuprl.org/html/02cucs-NuprlManual.pdf`.
28. Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
29. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 42–54. ACM, 2006.
30. Guodong Li. *Formal Verification of Programs and Their Transformations*. PhD thesis, University of Utah, 2010.
31. John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM*, 3(4):184–195, 1960.
32. P.F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1988.
33. Vincent Rahli, Nicolas Schiper, Robbert Van Renesse, Mark Bickford, and Robert L. Constable. A diversified and correct-by-construction broadcast service. In *The 2nd Int'l Workshop on Rigorous Protocol Engineering (WRiPE)*, Austin, TX, October 2012.
34. Nicolas Schiper, Vincent Rahli, Robbert Van Renesse, Mark Bickford, and Robert L. Constable. ShadowDB: A replicated database on a synthesized consensus core. In *Eighth Workshop on Hot Topics in System Dependability*, HotDep'12, 2012.
35. Scott F. Smith. *Partial Objects in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1989.