

# 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 [16,28,4] to reason about its own computation system and programming language, an applied lazy (call-by-name)  $\lambda$ -calculus. Since the computation system is universal (Turing complete), we need to reason using *partial types* introduced by Constable and Smith [38,17] and extended by Crary [18].<sup>1</sup> The *bisimulation* relation defined by Howe turned out to form a contextual equivalence relation [24,25], 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 [38,17,18,22,26,27]. 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

---

<sup>1</sup> 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 [11,34], 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 [29], and observed a significant speed-up. These synthesized consensus protocols have successfully been used in a replicated database [35]. 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 [23]. 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  $\lambda$ -calculus. Fig. 1 introduces a subset of this language, where  $\underline{n}$  ranges over integers. Because this language

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

Fig. 1: Syntax of Nuprl's programming language

is lazy, its values<sup>2</sup> (or canonical forms) are either integers, lambda abstractions, pairs, injections, or  $\text{Ax}$ . The canonical form  $\text{Ax}$  (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  $\text{ispair}$ . Adding these primitive forms was a design decision we made to distinguish between canonical forms (e.g., see  $\text{list\_ind}$ '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  $\perp$  (bottom) be  $\text{fix}(\lambda x.x)$ , let  $\pi_1(t)$  be  $(\text{let } x, y = t \text{ in } x)$ , and let  $\pi_2(t)$  be  $(\text{let } x, y = t \text{ in } y)$ .

Free and bound variables are defined as usual. We write  $t[x \setminus u]$  (and more generally  $t[x_1 \setminus u_1; \dots; x_n \setminus u_n]$ ) 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  $\text{Top}$  be the following type: for all closed terms  $t_1$  and  $t_2$ ,  $t_1 =_{\text{Top}} t_2$ .  $\text{Top}$ '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,  $T \sqsubseteq \text{Top}$  for every type  $T$ . Sec. 3.1 discusses the type  $\text{Base}$ , which

<sup>2</sup> 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:			
$(\lambda x.F) a$			$\rightarrow F[x \setminus a]$
$\text{let } x, y = \langle t_1, t_2 \rangle \text{ in } F$			$\rightarrow F[x \setminus t_1; y \setminus t_2]$
$\text{case inl}(t) \text{ of inl}(x) \Rightarrow F \mid \text{inr}(y) \Rightarrow G$			$\rightarrow F[x \setminus t]$
$\text{case inr}(t) \text{ of inl}(x) \Rightarrow F \mid \text{inr}(y) \Rightarrow G$			$\rightarrow G[y \setminus t]$
$\text{if } \underline{n}_1 = \underline{n}_2 \text{ then } t_1 \text{ else } t_2$			$\rightarrow t_1, \text{ if } \underline{n}_1 = \underline{n}_2$
$\text{if } \underline{n}_1 = \underline{n}_2 \text{ then } t_1 \text{ else } t_2$			$\rightarrow t_2, \text{ if } \underline{n}_1 \neq \underline{n}_2$
$\text{fix}(t)$			$\rightarrow t \text{ fix}(t)$
$\text{let } x := t_1 \text{ in } t_2$			$\rightarrow t_2[x \setminus t_1], \text{ if } t_1 \text{ is a value}$
Canonical form tests:			
$\text{ispair}(\langle t, t' \rangle, t_1, t_2) \rightarrow t_1$	$\text{ispair}(v, t_1, t_2) \rightarrow t_2, \text{ if } v \text{ is not a pair}$		
$\text{isaxiom}(\mathbf{Ax}, t_1, t_2) \rightarrow t_1$	$\text{isaxiom}(v, t_1, t_2) \rightarrow t_2, \text{ if } v \text{ is not axiom}$		
$\text{islambd}(\lambda x.t, t_1, t_2) \rightarrow t_1$	$\text{islambd}(v, t_1, t_2) \rightarrow t_2, \text{ if } v \text{ is not a lambda}$		
$\text{isinl}(\text{inl}(t), t_1, t_2) \rightarrow t_1$	$\text{isinl}(v, t_1, t_2) \rightarrow t_2, \text{ if } v \text{ is not a left injection}$		
$\text{isinr}(\text{inr}(t), t_1, t_2) \rightarrow t_1$	$\text{isinr}(v, t_1, t_2) \rightarrow t_2, \text{ if } v \text{ is not a right injection}$		
$\text{isint}(\underline{n}, t_1, t_2) \rightarrow t_1$	$\text{isint}(v, t_1, t_2) \rightarrow t_2, \text{ if } v \text{ is not an integer}$		

---

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  $\mathbf{Base}$ <sup>3</sup>) because equality on  $\mathbf{Base}$  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.<sup>4</sup>

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

<sup>3</sup> Being extentional, function types are in general not subtypes of  $\mathbf{Base}$ .

<sup>4</sup> 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 Appendix D), 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:  $\mathbb{B} = \mathbf{Unit} + \mathbf{Unit}$ . The  $\mathbf{Unit}$  type is defined as  $0 =_{\mathbb{Z}} 0$  and therefore  $\mathbf{Ax}$  is its only inhabitant (up to computation). We define the Boolean true  $\mathbf{tt}$  as  $\mathbf{inl}(\mathbf{Ax})$ , and the Boolean false  $\mathbf{ff}$  as  $\mathbf{inr}(\mathbf{Ax})$ . Using the *decide* operator we define a conditional operator as follows:  $\mathbf{if } t_1 \mathbf{ then } t_2 \mathbf{ else } t_3 = \mathbf{case } t_1 \mathbf{ of } \mathbf{inl}(x) \Rightarrow t_2 \mid \mathbf{inr}(x) \Rightarrow t_3$ .

**Lists** We define lists as follows using *Nuprl*'s union type [27] and recursive type [33] that allows one to build inductive types:<sup>5</sup>  $\mathbf{List}(T) = \mathbf{rec}(L.\mathbf{Unit} \cup T \times L)$ . 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  $\mathbf{Unit}$ , i.e.,  $\mathbf{Ax}$ , or a pair. The empty list  $\mathbf{nil}$  is defined as  $\mathbf{Ax}$ , and the cons operation, denoted by  $\bullet$ , as the pair constructor. We can distinguish an empty list and a non empty list because  $\mathbf{Unit}$  and the product type are disjoint. Using  $\mathbf{fix}$ , we define the following “list induction” operator:

$$\begin{aligned} \mathbf{list\_ind}(L, b, f) = \\ \mathbf{fix}(\lambda F.\lambda L.\mathbf{ispair}(L, \mathbf{let } h, t = L \mathbf{ in } f \ h \ t \ (F \ t), \mathbf{isaxiom}(L, b, \perp))) \ L \end{aligned}$$

To define such a function that takes a list as input, we need to be able to test whether it is a pair or  $\mathbf{Ax}$ . If we were to use the *spread* operator, we could destruct pairs, but computations would get stuck on  $\mathbf{Ax}$  which we use to represent the empty list. Therefore, we need an operator such as the  $\mathbf{ispair}$  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  $\mathbf{list\_ind}$ 's first argument does not compute to a pair or to  $\mathbf{Ax}$ , 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:

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

## 3 Computational Equivalence

### 3.1 Simulations and Bisimulations

Howe [24,25] 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  $\Theta(u_1, \dots, u_n)$  of the language defined in Sec. 2.1, then  $t_2$  computes to a canonical form  $\Theta(u'_1, \dots, u'_n)$  such that for all  $i \in \{1, \dots, n\}$ ,  $u_i \leq u'_i$ ). 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.<sup>6</sup>

<sup>5</sup> 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.

<sup>6</sup> 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*. Appendix D 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:

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

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

Let us write  $\mathbf{halts}(t)$  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  $(\mathbf{let } x := t_1 \mathbf{ in } t_2)$  first evaluates  $t_1$ . The term  $t_1$  converges if and only if the term  $(\mathbf{let } x := t_1 \mathbf{ in } \mathbf{Ax})$  evaluates to  $\mathbf{Ax}$ . So we simply define  $\mathbf{halts}(t)$  to be the simulation  $\mathbf{Ax} \leq (\mathbf{let } x := t \mathbf{ in } \mathbf{Ax})$ . Because  $\mathbf{Ax}$  is a canonical value then  $\mathbf{Ax} \leq (\mathbf{let } x := t \mathbf{ in } \mathbf{Ax})$  is true if and only if  $(\mathbf{let } x := t \mathbf{ in } \mathbf{Ax})$  computes to  $\mathbf{Ax}$ , i.e., if and only if  $t$  computes to a value.

Constable and Smith [38,17] 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  $\mathbf{Base} = \overline{\mathbf{Value}}$  where  $\mathbf{Value}$  is the type of all closed canonical terms of the computation system with  $\sim$  as its equality. Because  $\mathbf{Base}$  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  $\mathbf{Top}$ ,  $\mathbf{map}(f, t) @ \mathbf{nil} \sim \mathbf{map}(f, t)$ .

If  $t$  is a list, the first expression  $(\mathbf{map}(f, t) @ \mathbf{nil})$  requires two passes over the list  $t$  while the second expression  $(\mathbf{map}(f, t))$  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  $\mathbf{Top}$ . 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  $\mathbf{Nuprl}$  is based on an extension of the *untyped*  $\lambda$ -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  $t$ ,  $t @ \text{nil} \sim t$ . For example, by definition of  $@$ ,  $(\lambda x.x) @ \text{nil} \sim \perp$ . However, the bisimulation  $\lambda x.x \sim \perp$  is not true because the simulation  $\lambda x.x \leq \perp$  is not true. This shows that there are some terms  $t$  for which  $t \leq t @ \text{nil}$  does not hold.<sup>7</sup> However, Sec. 4 proves that for all terms  $t$ ,  $t @ \text{nil} \leq t$ . A corollary of that lemma is that  $\text{map}(f, t) @ \text{nil} \leq \text{map}(f, t)$ .

## 4 Proof Techniques

This section presents three proof techniques we use to prove bisimulations: Cray'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  $\text{map}(f, t) @ \text{nil} \sim \text{map}(f, t)$ , and in Sec. 5, we optimize distributed processes.

### 4.1 Least Upper Bound Property

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

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

### 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.<sup>8</sup> 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  $\text{ispair}(t_1, t_2, t_3) \in T$ , it is enough to prove  $\text{halts}(t_1)$ , 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  $\text{halts}(t)$  then  $\text{ispair}(t, \text{tt}, \text{ff}) \in \mathbb{B}$ .*

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  $\text{ispair}(t_1, t_2, t_3)$  always computes to  $t_2$  and cannot compute to  $t_3$  then we know that  $t_1$  is a pair. If we know that  $\text{ispair}(t_1, t_2, t_3)$  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:

<sup>7</sup> Appendix A provides a characterization of the terms that satisfy that property.

<sup>8</sup> 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 Appendix E.



---

$\vdash \forall F:\text{Top}. (\text{let } x, y = Ax \text{ in } F[x; y] \sim \text{bottom}())$
$\mid$
BY (SqReasoning
THEN Assert $\lceil (\text{if } Ax \text{ is a pair then } 0 \text{ otherwise } 1) = 1 \rceil$ .
THEN (Reduce 0 THEN Auto THEN AutoPairEta [2;1] (-1))

---

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

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

**Rule [not-ispair].** *To prove  $\text{ispair}(t_1, t_2, t_3) \sim t_3$ , it is enough to prove that  $\text{ispair}(t_1, \text{inl}(a), \text{inr}(b)) \sim \text{inr}(b)$  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  $\text{halts}(t)$  then  $t \sim \langle \pi_1(t), \pi_2(t) \rangle \vee \text{ispair}(t, a, b) \sim b$ .*

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

### 4.3 Convergence

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

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

Nuprl also has rules to reason about  $\text{halts}(t)$ . 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  $\text{halts}(\text{let } x, y = p \text{ in } F)$  then  $p$  computes to a pair.*

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

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,  $\text{let } x, y = Ax \text{ in } F \sim \perp$*

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



#### 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:*

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

*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.,  $\text{let } c, d = (\text{let } a, b = p \text{ in } F) \text{ in } G \leq \text{let } a, b = p \text{ in } (\text{let } c, d = F \text{ in } G)$ . Using [convergence], we can assume  $\text{halts}(\text{let } c, d = (\text{let } a, b = p \text{ in } F) \text{ in } G)$ , 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  $\text{let } c, d = F[a \setminus \pi_1(p); b \setminus \pi_2(p)] \text{ in } G \leq \text{let } c, d = F[a \setminus \pi_1(p); b \setminus \pi_2(p)] \text{ in } G$ , which is true by reflexivity of  $\leq$ .  $\square$

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

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

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:*

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

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:*

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

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

#### 4.6 Strictness

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

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

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  $\perp$ , then the computation diverges. Therefore, the entire term is computationally equal to  $\perp$ .

#### 4.7 Back To Our List Example

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

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

*Proof.* Because  $@$  is defined using  $\text{fix}$  (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

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

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

$$F^n \perp t \leq t$$

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

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

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

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:

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

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:  $((F^{n-1} \perp) \pi_2(t)) \leq \pi_2(t)$ .

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

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

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

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

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

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

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

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

$$\begin{aligned} & \text{ispair}(t, \text{let } x, y = t \text{ in } (f x) \bullet ((F^{n-1} \perp) y), \text{isaxiom}(t, \text{nil}, \perp)) & (2) \\ & \leq \text{map}(f, t) @ \text{nil} \end{aligned}$$

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

$$\begin{aligned} & \text{map}(f, t) @ \text{nil} \sim P @ \text{nil} \\ & \sim \text{ispair}(P, \text{let } x, y = t \text{ in } x \bullet (y @ \text{nil}), \text{isaxiom}(P, \text{nil}, \perp)) \end{aligned}$$

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

$$\begin{aligned} & \text{map}(f, t) @ \text{nil} \\ & \sim \text{ispair}(t, \text{let } x, y = t \text{ in } (f x) \bullet (\text{map}(f, y) @ \text{nil}), \text{isaxiom}(t, \text{nil}, \perp)) \end{aligned}$$

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

$$\begin{aligned} & \text{ispair}(t, \text{let } x, y = t \text{ in } (f x) \bullet ((F^{n-1} \perp) y), \text{isaxiom}(t, \text{nil}, \perp)) \\ & \leq \text{ispair}(t, \text{let } x, y = t \text{ in } (f x) \bullet (\text{map}(f, y) @ \text{nil}), \text{isaxiom}(t, \text{nil}, \perp)) \end{aligned}$$

which is true by induction hypothesis and the context property.  $\square$

## 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 [35]. 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:

$$\text{corec}(\lambda P.A \rightarrow P \times \text{Bag}(B))$$

where `corec` is defined as follows:

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

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  $b \gg= f = \text{bconcat}(\text{bmap}(f, b))$ . For example,  $(\{1; 2; 2; 4\} \gg= \lambda x.\{x; x + 1\}) = \{1; 2; 2; 3; 2; 3; 4; 5\} = \{1; 2; 2; 2; 3; 3; 4; 5\}$ .

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 `fix(λF.λs.λm.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:

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

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  $X$ :

$$f \circ X = \text{fix}(\lambda F. \lambda X. \lambda m. \left( \begin{array}{l} \text{let } X', out = X \ m \ \text{in} \\ \text{let } out' ::= \text{bmap}(f, out) \ \text{in} \\ \langle F \ X', out' \rangle \end{array} \right) X)$$

The state maintained by  $f \circ X$  is the state maintained by  $X$ . Note that for efficiency issues, we use the call-by-value all 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:

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

The state maintained by  $\text{buffer}(X, init)$  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:

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

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  $\text{fix}$ . We will now show that  $P$  is computationally equivalent to the following even simpler process:

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

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:

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

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  $\text{process}(n, L, S, R, I)$ .

**Lemma 10.** *Given a term  $f$  of type  $\text{Top}$ , the following bisimulation is true:*

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

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

$$\begin{aligned} & f \circ \text{process}(n, L, S, R, I) \\ & \sim \text{process}(n + 1, \\ & \quad \lambda s. \lambda m. \lambda i. \text{if } i = n + 1 \text{ then } \lambda x_1 \dots \lambda x_n. \text{bmap}(f, R \ s \ m \ x_1 \ \dots \ x_n), \\ & \quad \quad \quad \text{else } L \ s \ m \\ & \quad \lambda s. \lambda m. \lambda x_1 \dots \lambda x_n. \lambda x_{n+1}. S \ s \ m \ x_1 \ \dots \ x_n, \\ & \quad \lambda s. \lambda m. \lambda x_1 \dots \lambda x_n. \lambda x_{n+1}. x_{n+1}, \\ & \quad I) \end{aligned}$$

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

$$\begin{aligned} & \text{buffer}(\text{process}(n, L, S, R, I), I') \\ & \sim \text{process}(n + 1, \\ & \quad \lambda s. \lambda m. \lambda i. \text{if } i = n + 1 \\ & \quad \quad \quad \text{then } \lambda x_1 \dots \lambda x_n. (R \ \pi_1(s) \ m \ x_1 \ \dots \ x_n) \gg= \lambda f. (\pi_2(s) \gg= f), \\ & \quad \quad \quad \text{else } L \ \pi_1(s) \ m \\ & \quad \lambda s. \lambda m. \lambda x_1 \dots \lambda x_n. \lambda x_{n+1}. \langle S \ \pi_1(s) \ m \ x_1 \ \dots \ x_n \\ & \quad \quad \quad , \text{if } \text{bnull}(x_{n+1}) \text{ then } s \ \text{else } x_{n+1} \rangle, \\ & \quad \lambda s. \lambda m. \lambda x_1 \dots \lambda x_n. \lambda x_{n+1}. x_{n+1}, \\ & \quad \langle I, I' \rangle) \end{aligned}$$

**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  $\text{Top}$ , and all closed terms are trivially in  $\text{Top}$ .

## 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 [32] 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  $\text{iterate}(f, f \ x)$

and  $\text{map}(f, \text{iterate}(f, x))$  are bisimilar, where  $\text{iterate}(f, x)$  is defined in `Nuprl` as  $\text{fix}(\lambda F. \lambda x. \langle x, F(f\ x) \rangle) x$ . `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 [30]. 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” [30]. 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 [31] 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. Carl A. Gunter and Dana S. Scott. Semantic domains. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pages 633–674. 1990.
22. 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.
23. Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.
24. 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.
25. Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Inf. Comput.*, 124(2):103–112, 1996.
26. Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *LICS*, pages 86–95. IEEE Computer Society, 2003.
27. Alexei Kopylov. *Type Theoretical Foundations for Data Structures, Classes, and Objects*. PhD thesis, Cornell University, Ithaca, NY, 2004.
28. 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](http://www.nuprl.org/html/02cucs-NuprlManual.pdf).
29. Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
30. 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.
31. Guodong Li. *Formal Verification of Programs and Their Transformations*. PhD thesis, University of Utah, 2010.
32. John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM*, 3(4):184–195, 1960.
33. P.F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1988.
34. 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.

April 22, 2013

35. 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.
36. Dana S. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3):522–587, 1976.
37. Dana S. Scott. Logic and programming languages. *Commun. ACM*, 20(9):634–641, 1977.
38. Scott F. Smith. *Partial Objects in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1989.

## A Remark About $u @ \text{nil} \sim u$

Note that  $u @ \text{nil} \sim u$  is not only true for  $u = \text{map}(f, t)$  but for a larger class of terms. Therefore, an obvious question is, how can we characterize that class of terms? It turns out that these terms are the ones that satisfy the following property which we call  $P$ : if they have a value then they are either  $\text{Ax}$  or are pairs such that this property applies recursively on their second components. Let us present two such characterizations. One of them is more “syntactical” ( $\text{ListLike1}$ ) and the other one more “semantical” ( $\text{ListLike2}$ ). Both lists and streams satisfy these properties.

We can formally define  $P$  as the following proposition:

$$\text{ListLike1}(t) = \bigcap n : \mathbb{N}. \text{ListLike1N}(t, n)$$

where  $\text{ListLike1N}$  is defined as follows:

$$\begin{aligned} & \text{ListLike1N}(t, n) \\ = & \text{fix} \left( \begin{array}{l} \lambda F. \lambda n. \text{if } n = 0 \text{ then } \lambda t. \text{Unit} \\ \text{else } \lambda t. \bigcap x : \text{halts}(t). \text{ispair} \left( \begin{array}{l} t, \\ F \pi_2(t), \\ \text{isaxiom}(t, \text{tt}, \text{ff}) \end{array} \right) \end{array} \right) n t \end{aligned}$$

We have proved that for all terms  $t$  in  $\text{Base}$ ,  $\text{ListLike1}(t)$  iff  $t @ \text{nil} \sim t$ ; and we have proved, among other things, that for all terms  $t$  and  $f$  in  $\text{Base}$ ,  $\text{ListLike1}(\text{map}(f, t))$  is true.

We can also formally define  $P$  as the following proposition:

$$\text{ListLike2} = \text{corec}(\lambda T. \overline{\text{Unit} \cup \text{Top} \times T})$$

We have proved that for all terms  $t$  in  $\text{Base}$ ,  $t \in \text{ListLike2}$  iff  $t @ \text{nil} \sim t$ ; and we have proved, among other things, that for all terms  $t$  and  $f$  in  $\text{Base}$ ,  $\text{map}(f, t) \in \text{ListLike2}$ .

## B Connection to Domain Theory

Our approach is quite similar to the Domain Theoretic techniques of reasoning about programs [36,37,21]. The members of  $\text{Base}$ , along with the relation  $\leq$ , form a partial order. The least upper bound property discussed in Sec. 4.1 essentially says that this partial order is complete for chains of a particular form. Even though it does not explicitly say that all chains have a least upper bound in  $\text{Base}$ , it is useful in many proofs.

## C Constructive Proof of Crary’s Compactness Property

In addition to the least upper bound property presented in Sec. 4.1, Crary [18] proves the following compactness property: for all terms  $G$  and  $f$ , if  $G(\text{fix}(f))$  converges, then there exists a natural number  $n$  such that  $\text{halts}(G(f^n(\perp)))$ .

Using compactness we can characterize lists using the following recursive function that checks whether or not a term is a list, and diverges if it is not:

$$\text{islist}(t) = \text{fix}(\lambda F. \lambda t. \text{ispair}(t, F \pi_2(t), \text{isaxiom}(t, \text{tt}, \perp))) t$$

We have proved the following results:

**Lemma 13.** *For all terms  $t$  in  $\text{Base}$ , if  $\text{halts}(\text{islist}(t))$  then  $t \in \text{List}(\text{Base})$ .*

We prove that result using compactness since  $\text{halts}(\text{islist}(t))$  is of the form  $\text{halts}(G(\text{fix}(f)))$ .

**Lemma 14.** *For all type  $T$  and for all terms  $t$  in  $\text{List}(T)$ ,  $\text{halts}(\text{islist}(t))$ .*

We prove that result by induction on the list  $t$ .

Crary gives a non-constructive proof of the compactness property in his thesis. He also provides enough hints for deriving a constructive proof. For the sake of completeness, we formally present the constructive proof here. Informally, this lemma says that if  $\text{fix}(f)$  converges, then there is a number  $n$  such that the  $n^{\text{th}}$  approximation to  $\text{fix}(f)$  converges, i.e.,  $f^n(\perp)$  converges. For brevity, we will denote  $f^n(\perp)$  by  $f^n$ . This proof is a direct consequence of his Lemma 5.8. We restate it here for the reader's convenience. It's constructive proof can be found in the appendix of his thesis [18].

**Lemma 15 (Crary [18, Lemma 5.8]).** *For all  $f$ ,  $e_1$ ,  $e_2$  (where  $f$  is closed and  $x$  is the only free variable of  $e_1$ ), if  $e_1[x \setminus \text{fix}(f)] \rightarrow^* e_2$ , then there exists  $j$  and  $e'_2$  such that  $e_2 \equiv e'_2[x \setminus \text{fix}(f)]$  and for all  $k \geq j$ ,  $e'_2[x \setminus f^{k-j}] \leq e_1[x \setminus f^k]$ .*

We write  $t_1 \equiv t_2$  to mean that  $t_1$  and  $t_2$  are alpha-equal terms. Now, we provide a meta-theoretical proof of compactness. Suppose  $G(\text{fix}(f))$  converges. Let  $v$  be the canonical value to which it converges.<sup>9</sup> Since  $G$  is the principal argument, we know that  $G \rightarrow^* \lambda x.b$  for term  $b$ . One more reduction step gives us  $G(\text{fix}(f)) \rightarrow^* (\lambda x.b) \text{fix}(f) \rightarrow^1 b[x \setminus \text{fix}(f)] \rightarrow^* v$ . We can now invoke Lemma 15 with  $e_1$  as  $b$  and  $e_2$  as  $v$ . We get a number  $j$  and a term  $e'_2$  such that  $v \equiv e'_2[x \setminus \text{fix}(f)]$  and for all  $k \geq j$ ,  $e'_2[x \setminus f^{k-j}] \leq b[x \setminus f^k]$ . In particular let us chose  $k = j$ . By definition,  $f^0$  is  $\perp$ . So, we have  $e'_2[x \setminus \perp] \leq b[x \setminus f^j]$ . Because  $v$  is a value,  $e'_2[x \setminus \text{fix}(f)]$  and  $e'_2[x \setminus \perp]$  must be values too. So, by definition of  $\leq$ ,  $b[x \setminus f^j]$  converges to a value with same outer canonical operator as  $v$ . Because  $G(f^j) \rightarrow^* b[x \setminus f^j]$ ,  $G(f^j)$  also converges. This completes the proof of compactness (with  $n$  as  $j$ ).

## D Extensionality of the New Operators

Howe [24] showed that to ensure that  $\leq$  and hence  $\sim$  is a congruence, we need to make sure that all the operators satisfies an *extensionality* condition. This section sketches meta-theoretical proofs that the new operators introduced in this paper ( $\text{fix}$ , call-by-value, call-by-valueall, and the canonical form tests such as  $\text{ispair}$ ) are extensional.

First, let us present some relations introduced by Howe [24]: let  $u \rightarrow^k b$  be true if  $u$  reduces to  $b$  in  $k$  steps.<sup>10</sup> let  $u \rightarrow^{<k} b$  be true if  $u$  reduces to  $b$  in strictly less than  $k$  steps. let  $\leq^*$  be an extension of  $\leq$  which satisfies congruence

<sup>9</sup> In meta-theory, we have access to the entire sequence of reductions from  $G(\text{fix}(f))$  to  $v$ .

<sup>10</sup> This definition contrasts with Howe's notation [24] where  $b$  is required to be a value.

by definition. Howe writes: “Informally,  $a \leq^* b$  if  $b$  can be obtained from  $a$  via one bottom-up pass of replacements of subterms by terms that are larger under  $\leq$ .” Howe mentions that the following holds: if  $a \leq^* b$  and  $b \leq c$  then  $a \leq^* c$ .

Note that in a deterministic computation system like ours, if  $u \rightarrow v$  or  $u \rightarrow^k v$ , or  $u \rightarrow^{<k} v$ , then  $u \leq v$  and  $v \leq u$ .

Howe defines the extensionality condition on operators as follows [24, Def. 5]: An operator  $\tau$  is extensional if for any closed term  $\tau(t_1, \dots, t_n)$ ,  $\tau(t'_1, \dots, t'_n)$  and  $a$ , and for any  $k \geq 0$ , if

1.  $\tau(t_1, \dots, t_n) \rightarrow^k a$ ,
2.  $\tau(t_1, \dots, t_n) \leq^* \tau(t'_1, \dots, t'_n)$  and
3. for every closed  $u, u'$  and  $v$ , if  $u \rightarrow^{<k} u'$  and  $u \leq^* v$  then  $u' \leq^* v$ ,

then  $a \leq^* \tau(t'_1, \dots, t'_n)$ .

The fixpoint operator `fix` is extensional because it can be defined as follows:

$$(\lambda x.f (x x)) (\lambda x.f (x x))$$

The call-by-value operator is also extensional because it can be defined as follows:

$$\text{fix} \left( \lambda F.\lambda t. \left( \text{let } x := t \text{ in } \left( \begin{array}{l} x, \\ \text{let } a, b = x \text{ in let } c := F a \text{ in let } d := F b \text{ in } \langle c, d \rangle, \\ \text{ispair} \left( \begin{array}{l} x, \\ \text{let } y := F \text{ outl}(x) \text{ in inl}(y), \\ \text{isinl} \left( \begin{array}{l} x, \\ \text{isinr} \left( \text{let } y := F \text{ outr}(x) \text{ in inr}(y), \right) \right) \right) \end{array} \right) \right) \right) \right) t$$

**Lemma 16.** *The call-by-value operator is extensional.*

*Proof.* Let  $\text{let } x := t_1 \text{ in } t_2$ ,  $\text{let } x := t'_1 \text{ in } t'_2$ , and  $a$  be closed terms. Let us assume:

1.  $\text{let } x := t_1 \text{ in } t_2 \rightarrow^k a$  and  $a$  is a value.
2.  $t_1 \leq^* t'_1$ , and  $t_2 \leq^* t'_2$
3. for every closed  $u, u'$ , and  $v$ , if  $u \rightarrow^{<k} u'$  and  $u \leq^* v$  then  $u' \leq^* v$

We have to prove  $a \leq^* \text{let } x := t'_1 \text{ in } t'_2$ . Using a variant of [halt-spread] for the call-by-value operator, and assumption 1,  $t_1$  must compute to a value  $b$  of the form  $\Theta(u_1, \dots, u_n)$ . We then have the following sequence of reductions:

$$\text{let } x := t_1 \text{ in } t_2 \rightarrow^{<k} \text{let } x := b \text{ in } t_2 \rightarrow t_2[x \setminus b] \rightarrow^{<k} a$$

Because  $t_1 \rightarrow^{<k} b$  and because by assumption 2,  $t_1 \leq^* t'_1$ , then by assumptions 3, we have  $b \leq^* t'_1$ . Using Howe’s Lemma 2 [24], we have  $t'_1 \rightarrow^{<l} b'$ , for some natural number  $l$ , and some closed terms  $b'$  is of the form  $\Theta(u'_1, \dots, u'_n)$  such that for all  $i \in \{1, \dots, n\}$ ,  $u_i \leq^* u'_i$ . So we have  $\text{let } x := t'_1 \text{ in } t'_2 \rightarrow^{<l+1}$

$t'_2[x \setminus b']$ . Because  $\text{let } x := t'_1 \text{ in } t'_2 \rightarrow^{<l+1} t'_2[x \setminus b']$ , we obtain  $t'_2[x \setminus b'] \leq \text{let } x := t'_1 \text{ in } t'_2$ . Assumption 2 says  $t_2 \leq^* t'_2$  and we have  $b \leq^* b'$ , so using Howe's Lemma 1 [24], we obtain  $t_2[x \setminus b] \leq^* t'_2[x \setminus b']$ . Therefore, we have  $t_2[x \setminus b] \leq^* \text{let } x := t'_1 \text{ in } t'_2$ . Finally, using Assumption 3, because  $t_2[x \setminus b] \rightarrow^{<k} a$ , we obtain  $a \leq^* \text{let } x := t'_1 \text{ in } t'_2$ .  $\square$

**Lemma 17.** *The `ispair` operator is extensional.*

*Proof.* To prove this requirement, we can assume the following for closed terms  $t, t_1, t_2, t', t'_1, t'_2$ :

1.  $\text{ispair}(t, t_1, t_2) \rightarrow^k a$  and  $a$  is a value.
2.  $t \leq^* t', t_1 \leq^* t'_1$ , and  $t_2 \leq^* t'_2$
3. for every closed  $u, u'$ , and  $v$ , if  $u \rightarrow^{<k} u'$  and  $u \leq^* v$  then  $u' \leq^* v$

We have to prove  $a \leq^* \text{ispair}(t', t'_1, t'_2)$ . Using `[halt-ispair]` and assumption 1,  $t$  must converge to a value  $b$ . We can decide whether  $b$  is a pair or some other canonical value (by checking  $b$ 's outer canonical operator). Depending on whether or not  $b$  is a pair, either of the two following cases can happen:

1.  $\text{ispair}(t, t_1, t_2) \rightarrow^{<k} \text{ispair}(b, t_1, t_2) \rightarrow t_1 \rightarrow^{<k} a$ , if  $b$  is a pair
2.  $\text{ispair}(t, t_1, t_2) \rightarrow^{<k} \text{ispair}(b, t_1, t_2) \rightarrow t_2 \rightarrow^{<k} a$ , if  $b$  is not a pair

We consider case 1. The other case is similar. In this case,  $b$  is of the form  $\langle c, d \rangle$  for some closed terms  $c, d$ . Because  $t$  reduces to  $\langle c, d \rangle$  in less than  $k$  steps, and because by assumption 2,  $t \leq^* t'$ , then by assumption 3 we have  $\langle c, d \rangle \leq^* t'$ . Using Howe's Lemma 2 [24], we have  $t' \rightarrow^{<l} \langle c', d' \rangle$ ,  $c \leq^* c'$  and  $d \leq^* d'$  for some natural number  $l$ , and some closed terms  $c'$  and  $d'$ . So we have  $\text{ispair}(t', t'_1, t'_2) \rightarrow^{<l+1} t'_1$ . Because  $\text{ispair}(t', t'_1, t'_2) \rightarrow^{<l+1} t'_1$ , we have  $t'_1 \leq \text{ispair}(t', t'_1, t'_2)$ . Assumption 2 says that  $t_1 \leq^* t'_1$ . Therefore, we obtain  $t_1 \leq^* \text{ispair}(t', t'_1, t'_2)$ . Finally, using assumption 3, and because  $t_1 \rightarrow^{<k} a$ , we obtain  $a \leq^* \text{ispair}(t', t'_1, t'_2)$ .  $\square$

## E Validity of the New Rules

This section shows that the rules discussed in the paper are all valid, and are therefore consistent with `Nuprl`. To prove that result, we use Allen's PER (Partial Equivalence Relations) semantic method [2,3]

The `Nuprl` programming/logical system can be presented in three phases. First, one defines the computation system, then one defines the type system, and finally one defines the proof rules.

Using Allen's method, one gives the semantics of types and of proof rules. The semantics of `Nuprl`'s types is characterized by ternary partial equivalence relations that express when two terms are equal in a type. In this paper we are especially interested in equality types of the form  $t_1 =_T t_2$ , simulation types of the form  $t_1 \leq t_2$ , and bisimulation types of the form  $t_1 \sim t_2$ .

`Nuprl`'s type theory relies on a hierarchy of universes `Type`<sub>1</sub>, `Type`<sub>2</sub>, `Type`<sub>3</sub>, etc., such that for all  $i \in \mathbb{N}^+$ , `Type` <sub>$i+1$</sub>  contains all types that can be built using

universes no higher than  $\mathbf{Type}_i$ . The type  $\mathbf{Type}_1$  contains all types that can be built without using universes. We write  $\mathbf{Type}$  or  $\mathbb{P}$  for  $\mathbf{Type}_i$  where  $i$  can be any member of  $\mathbb{N}^+$ .

Crary [18] introduced the simulation and bisimulation relations as meta-theoretical relations, and introduced an extra form of type judgment (sequent) to prove bisimulations. In this paper, simulations and bisimulations are also types of the logic and values of the computation system. They are inhabited by  $\mathbf{Ax}$  when the propositions they represent are true.

### E.1 Simulation and Bisimulation Types

Let us now extend Crary's type specifications and type definitions [18, Sec. 4.4] as follows: Two simulations are equal as types if they are both true or both false (i.e., the equality between simulation types is extensional):

$$t_1 \leq t_2 =_{\mathbf{Type}} u_1 \leq u_2 \iff (t_1 \leq t_2 \iff u_1 \leq u_2)$$

Similarly,

$$t_1 \sim t_2 =_{\mathbf{Type}} u_1 \sim u_2 \iff (t_1 \sim t_2 \iff u_1 \sim u_2)$$

The equality in simulation and bisimulation types can be characterized as follows:

$$\mathbf{Ax} \in t_1 \leq t_2 \iff t_1 \leq t_2 \in \mathbf{Type} \wedge t_1 \leq t_2$$

$$\mathbf{Ax} \in t_1 \sim t_2 \iff t_1 \sim t_2 \in \mathbf{Type} \wedge t_1 \sim t_2$$

We characterize the equality between equality types of the form  $t_1 =_T t_2$ , and the equality in equality types as follows:

$$t_1 =_T t_2 =_{\mathbf{Type}} u_1 =_U u_2 \iff T =_{\mathbf{Type}} U \wedge t_1 =_T u_1 \wedge t_2 =_T u_2$$

$$\mathbf{Ax} \in t_1 =_T t_2 \iff t_1 =_T t_2 \in \mathbf{Type} \wedge t_1 =_T t_2$$

The next step is to define for each type constructor an inductive relation that satisfies the specification of the type. These relations can then be combined into one to form a type system. Crary [18, Def. 4.9] defines such a relation and proves that it adheres to the corresponding type specification he provides [18, Fig. 4.5]. That relation can be expressed by two following relations:  $T_1 = T_2$  that expresses which terms are types and which are equal types, and  $t_1 =_{t_2} \in T$  that expresses which terms belong to a type and which are equal within that type. We have informally checked that Crary's proof can be extended with relations that satisfy the specifications of simulation and bisimulation types. We leave the formal extension of Crary's proof for future work.

### E.2 Hypotheses, Sequents, and Proof Rules

Nuprl sequents are of the form  $H \vdash C$  [ $\mathbf{ext} \ t$ ], where  $H$  is a list of pairs variable/type of the form  $x_1 : T_1, \dots, x_n : T_n$  such that the variables  $x_1, \dots, x_n$  are all distinct from each other. Moreover, every free variable of  $T_i$  is one of  $x_1, \dots, x_{i-1}$  and every free variable of  $C$  or of  $t$  is one of  $x_1, \dots, x_n$ . The list  $H$



is called the hypothesis or assumption list of the sequent,  $C$  its conclusion, and  $t$  its extract or evidence.

A sequent of the form  $x_1 : T_1, \dots, x_n : T_n \vdash C$  [ext  $t$ ] roughly means that assuming that for all  $i \in \{1, \dots, n\}$ , if  $x_i$  has type  $T_i$ , then  $t$  is a member of type  $C$ . This is not exactly true because  $T_i$  might depend on  $x_1, \dots, x_{i-1}$ . We make that definition precise below.

We write  $H \vdash C$  for  $H \vdash C$  [ext  $\mathbf{Ax}$ ].

A rule is of the following form, where [name] is the name of the rule:

$$\begin{array}{c} H \vdash C \text{ [ext } t] \\ \text{BY [name]} \\ H_1 \vdash C_1 \text{ [ext } t_1] \\ \vdots \\ H_n \vdash C_n \text{ [ext } t_n] \end{array}$$

As usual, a rule is just an implication: if for all  $i \in \{1, \dots, n\}$ ,  $H_i \vdash C_i$  [ext  $t_i$ ] is a “true” sequent, then so is  $H \vdash C$  [ext  $t$ ].

In order to formally define what it means for a rule to be true, let us first repeat some useful relations on hypotheses and sequents.

First, let for any metavariable  $t$ , let  $t_{(n)}$  stand for the list  $t_1, \dots, t_n$ .

We say that a list of hypotheses  $H$  is true at a list of terms  $l$ , and write  $H @ l$ , if the following formula holds:

$$\begin{array}{l} x_1 : T_1, \dots, x_n : T_n @ t_{(n)} \\ \iff \\ \forall j < n. t_{j+1} \in T_{j+1}[t_{(j)}/x_{(j)}] \\ \quad \wedge \forall t'_{(j)}. \forall i < j. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}] \\ \quad \Rightarrow T_{j+1}[t_{(j)}/x_{(j)}] = T_{j+1}[t'_{(j)}/x_{(j)}] \end{array}$$

We say that a sequent of the form  $H \vdash C$  [ext  $t$ ] is true at a list of terms  $l$ , and write  $H \vdash C$  [ext  $t$ ] @  $l$ , if the following formula holds:

$$\begin{array}{l} x_1 : T_1, \dots, x_n : T_n \vdash C \text{ [ext } t] @ t_{(n)} \\ \iff \\ \forall t'_1, \dots, t'_j. \\ x_1 : T_1, \dots, x_n : T_n @ t_{(n)} \wedge \forall i < n. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}] \\ \Rightarrow C[t_{(n)}/x_{(n)}] = C[t'_{(n)}/x_{(n)}] \wedge t[t_{(n)}/x_{(n)}] = t[t'_{(n)}/x_{(n)}] \in C[t_{(n)}/x_{(n)}] \end{array}$$

Finally, a sequent of the form  $x_1 : T_1, \dots, x_n : T_n \vdash C$  [ext  $t$ ] is true iff  $\forall t_{(n)}. x_1 : T_1, \dots, x_n : T_n \vdash C$  [ext  $t$ ] @  $t_{(n)}$ .

We say that a rule of the form

$$\begin{array}{c} H \vdash C \text{ [ext } t] \\ \text{BY [name]} \\ H_1 \vdash C_1 \text{ [ext } t_1] \\ \vdots \\ H_n \vdash C_n \text{ [ext } t_n] \end{array}$$

is valid iff the truth of  $H_1 \vdash C_1$  [ext  $t_1$ ],  $\dots$ ,  $H_n \vdash C_n$  [ext  $t_n$ ] implies the truth of  $H \vdash C$  [ext  $t$ ].

---

$H \vdash G[\mathbf{fix}(f)/x] \leq t$ BY [least-upper-bound] $H, j : \mathbb{N} \vdash G[f^j(\perp)/x] \leq t$
---

---

Fig. 4: Least upper bound rule

---

$H \vdash a \leq a$ BY [sqle-refl]	$H \vdash a \sim a$ BY [sqequal-refl]	$H \vdash a \sim b$ BY [sqequal-base] $H \vdash a =_{\text{Base}} b$
$H \vdash a b \sim c d$ BY [sqequal-app-D] $H \vdash a \sim c$ $H \vdash b \sim d$	$H \vdash \lambda x.a \sim \lambda x.b$ BY [sqequal-lam-D] $H, x : \text{Base} \vdash a \sim b$	$H \vdash a \sim b$ BY [sqequal-sqle] $H \vdash a \leq b$ $H \vdash b \leq a$

---

Fig. 5: Simulation and bisimulation rules

---

$H \vdash \text{ispair}(a, b, c) \in T$ BY [ispair-member] $H \vdash \text{halts}(a)$ $H \vdash b \in T$ $H \vdash c \in T$ $H \vdash a \in \text{Base}$	$H \vdash \text{isaxiom}(a, b, c) \in T$ BY [isaxiom-member] $H \vdash \text{halts}(a)$ $H \vdash b \in T$ $H \vdash c \in T$ $H \vdash a \in \text{Base}$	$H \vdash \text{isinl}(a, b, c) \in T$ BY [isinl-member] $H \vdash \text{halts}(a)$ $H \vdash b \in T$ $H \vdash c \in T$ $H \vdash a \in \text{Base}$
$H \vdash \text{isinr}(a, b, c) \in T$ BY [isinr-member] $H \vdash \text{halts}(a)$ $H \vdash b \in T$ $H \vdash c \in T$ $H \vdash a \in \text{Base}$	$H \vdash \text{isint}(a, b, c) \in T$ BY [isint-member] $H \vdash \text{halts}(a)$ $H \vdash b \in T$ $H \vdash c \in T$ $H \vdash a \in \text{Base}$	$H \vdash \text{islambd}(a, b, c) \in T$ BY [islambd-member] $H \vdash \text{halts}(a)$ $H \vdash b \in T$ $H \vdash c \in T$ $H \vdash a \in \text{Base}$

---

Fig. 6: Canonical form test rules—membership rules

### E.3 Validity of the New Rules

Figures 4, 5, 6, 7, and 8 present the rules introduced in this paper.

**Lemma 18.** *[least-upper-bound] is valid*

*Proof.* Let us assume that the sequent  $H, j : \mathbb{N} \vdash G[f^j(\perp)/x] \leq t$  is true and let us prove that the sequent  $H \vdash G[\mathbf{fix}(f)/x] \leq t$  is true. Let  $H$  be  $x_1 : T_1, \dots, x_n : T_n$ . Let  $t_1, \dots, t_n, t'_1, \dots, t'_n$  be closed terms. Let us assume:

$$x_1 : T_1, \dots, x_n : T_n @ t_{(n)}$$

and

$$\forall i < n. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}]$$

We have to prove:

$$(G[\mathbf{fix}(f)/x] \leq t)[t_{(n)}/x_{(n)}] = (G[\mathbf{fix}(f)/x] \leq t)[t'_{(n)}/x_{(n)}] \quad (3)$$

and

$$\mathbf{Ax} = \mathbf{Ax} \in (G[\mathbf{fix}(f)/x] \leq t)[t_{(n)}/x_{(n)}] \quad (4)$$

Using our first hypothesis, we obtain that for all  $i \in \mathbb{N}$ :

$$(G[f^i(\perp)/x] \leq t)[t_{(n)}/x_{(n)}] = (G[f^i(\perp)/x] \leq t)[t'_{(n)}/x_{(n)}] \quad (5)$$

---

$H \vdash t \in \text{Top} \times \text{Top}$ BY [ispair] $H \vdash \text{ispair}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inl}(a)$	$H \vdash t \sim \text{Ax}$ BY [isaxiom] $H \vdash \text{isaxiom}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inl}(a)$
$H \vdash t \sim \text{inl}(\text{outl}(t))$ BY [isinl] $H \vdash \text{isinl}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inl}(a)$	$H \vdash t \sim \text{inr}(\text{outr}(t))$ BY [isinr] $H \vdash \text{isinr}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inl}(a)$
$H \vdash t \in \mathbb{Z}$ BY [isint] $H \vdash \text{isint}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inl}(a)$	$H \vdash t \sim \lambda v.(t \ v)$ BY [islambda] $H \vdash \text{islambda}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inl}(a)$
$H \vdash \text{ispair}(t, c, d) \sim d$ BY [not-ispair] $H \vdash \text{ispair}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inr}(b)$	$H \vdash \text{isaxiom}(t, c, d) \sim d$ BY [not-isaxiom] $H \vdash \text{isaxiom}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inr}(b)$
$H \vdash \text{isinl}(t, c, d) \sim d$ BY [not-isinl] $H \vdash \text{isinl}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inr}(b)$	$H \vdash \text{isinr}(t, c, d) \sim d$ BY [not-isinr] $H \vdash \text{isinr}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inr}(b)$
$H \vdash \text{isint}(t, c, d) \sim d$ BY [not-isint] $H \vdash \text{isint}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inr}(b)$	$H \vdash \text{islambda}(t, c, d) \sim d$ BY [not-islambda] $H \vdash \text{islambda}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inr}(b)$

---

Fig. 7: Canonical form test rules—semi-decision rules

---

$H \vdash t_1 \leq t_2$ BY [convergence] $H, y : \text{halts}(t_1) \vdash t_1 \leq t_2$ $H \vdash \text{halts}(t_1) \in \mathbb{P}$	
$H \vdash p \in \text{Top} \times \text{Top}$ BY [halt-spread] $H \vdash \text{halts}(\text{let } x, y = p \text{ in } F)$	$H \vdash d \in \text{Top} + \text{Top}$ BY [halt-decide] $H \vdash \text{halts}(\text{case } d \text{ of } \text{inl}(x) \Rightarrow F \mid \text{inr}(y) \Rightarrow G)$
$H \vdash f \sim \lambda v.(f \ v)$ BY [halt-apply] $H \vdash \text{halts}(f \ a)$	$H \vdash \text{halts}(t_1)$ BY [halt-callbyvalue] $H \vdash \text{halts}(\text{let } x := t_1 \text{ in } t_2)$
$H \vdash \text{halts}(t_1)$ BY [halt-ispair] $H \vdash \text{halts}(\text{ispair}(t_1, t_2, t_3))$	$H \vdash \text{halts}(t_1)$ BY [halt-isaxiom] $H \vdash \text{halts}(\text{isaxiom}(t_1, t_2, t_3))$
$H \vdash \text{halts}(t_1)$ BY [halt-isinl] $H \vdash \text{halts}(\text{isinl}(t_1, t_2, t_3))$	$H \vdash \text{halts}(t_1)$ BY [halt-isinr] $H \vdash \text{halts}(\text{isinr}(t_1, t_2, t_3))$
$H \vdash \text{halts}(t_1)$ BY [halt-isint] $H \vdash \text{halts}(\text{isint}(t_1, t_2, t_3))$	$H \vdash \text{halts}(t_1)$ BY [halt-islambda] $H \vdash \text{halts}(\text{islambda}(t_1, t_2, t_3))$

---

Fig. 8: Canonical form test rules—convergence rules

$$\text{Ax} = \text{Ax} \in (G[f^i(\perp)/x] \leq t)[t_{(n)}/x_{(n)}] \quad (6)$$

We prove 4 from 6 and using Crary's least upper bound theorem [18, Theorem 5.9]. To prove 3, we have to prove that  $(G[\mathbf{fix}(f)/x] \leq t)[t_{(n)}/x_{(n)}]$  if and only if  $(G[\mathbf{fix}(f)/x] \leq t)[t'_{(n)}/x_{(n)}]$ . Because we know that  $(G[\mathbf{fix}(f)/x] \leq t)[t_{(n)}/x_{(n)}]$  is true (from 4), it remains to prove that  $(G[\mathbf{fix}(f)/x] \leq t)[t'_{(n)}/x_{(n)}]$  is also true. Using Crary's least upper bound theorem [18, Theorem 5.9], it is enough to prove that for all  $i \in \mathbb{N}$ ,  $(G[f^i(\perp)/x] \leq t)[t'_{(n)}/x_{(n)}]$  is true. Using 5, it is equivalently enough to prove that for all  $i \in \mathbb{N}$ ,  $(G[f^i(\perp)/x] \leq t)[t_{(n)}/x_{(n)}]$  is true. We conclude using 6.  $\square$

**Lemma 19.** *[sqle-refl] is valid*

*Proof.* Let us prove that the sequent  $H \vdash a \leq a$  is true. Let  $H$  be  $x_1 : T_1, \dots, x_n : T_n$ . Let  $t_1, \dots, t_n, t'_1, \dots, t'_n$  be closed terms. Let us assume:

$$x_1 : T_1, \dots, x_n : T_n @ t_{(n)}$$

and

$$\forall i < n. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}]$$

We have to prove:

$$(a \leq a)[t_{(n)}/x_{(n)}] = (a \leq a)[t'_{(n)}/x_{(n)}]$$

and

$$\mathbf{Ax} = \mathbf{Ax} \in (a \leq a)[t_{(n)}/x_{(n)}]$$

which are both true by definition because  $\leq$  is reflexive.  $\square$

**Lemma 20.** *[sqequal-refl] is valid*

*Proof.* Let us prove that the sequent  $H \vdash a \sim a$  Let  $H$  be  $x_1 : T_1, \dots, x_n : T_n$ . Let  $t_1, \dots, t_n, t'_1, \dots, t'_n$  be closed terms. Let us assume:

$$x_1 : T_1, \dots, x_n : T_n @ t_{(n)}$$

and

$$\forall i < n. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}]$$

We have to prove:

$$(a \sim a)[t_{(n)}/x_{(n)}] = (a \sim a)[t'_{(n)}/x_{(n)}]$$

and

$$\mathbf{Ax} = \mathbf{Ax} \in (a \sim a)[t_{(n)}/x_{(n)}]$$

which are both true by definition because  $\sim$  is reflexive.  $\square$

**Lemma 21.** *[sqequal-base] is valid*

*Proof.* Let us assume that  $H \vdash a =_{\text{Base}} b$  is true and let us prove that  $H \vdash a \sim b$  is true. Let  $H$  be  $x_1 : T_1, \dots, x_n : T_n$ . Let  $t_1, \dots, t_n, t'_1, \dots, t'_n$  be closed terms. We assume that:

$$x_1 : T_1, \dots, x_n : T_n \text{ @ } t_{(n)}$$

and

$$\forall i < n. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}]$$

We have to prove:

$$(a \sim b)[t_{(n)}/x_{(n)}] = (a \sim b)[t'_{(n)}/x_{(n)}] \quad (7)$$

and

$$\mathbf{Ax} = \mathbf{Ax} \in (a \sim b)[t_{(n)}/x_{(n)}] \quad (8)$$

Using our hypothesis, we obtain that:

$$(a =_{\text{Base}} b)[t_{(n)}/x_{(n)}] = (a =_{\text{Base}} b)[t'_{(n)}/x_{(n)}]$$

and

$$\mathbf{Ax} = \mathbf{Ax} \in (a =_{\text{Base}} b)[t_{(n)}/x_{(n)}]$$

By definition of the equality in **Base** we obtain:

$$a[t_{(n)}/x_{(n)}] = a[t'_{(n)}/x_{(n)}] \in \text{Base} \quad (9)$$

$$b[t_{(n)}/x_{(n)}] = b[t'_{(n)}/x_{(n)}] \in \text{Base} \quad (10)$$

and

$$a[t_{(n)}/x_{(n)}] = b[t_{(n)}/x_{(n)}] \in \text{Base} \quad (11)$$

Therefore, using 9 and 10, we get that  $a[t_{(n)}/x_{(n)}] \sim a[t'_{(n)}/x_{(n)}]$  and  $b[t_{(n)}/x_{(n)}] \sim b[t'_{(n)}/x_{(n)}]$ . By transitivity of  $\sim$ , we obtain that 7 is true. From 11, we obtain that  $a[t_{(n)}/x_{(n)}] \sim b[t_{(n)}/x_{(n)}]$  and so 8 is also true  $\square$

**Lemma 22.** *[s<sub>equal</sub>-app-D] is valid*

*Proof.* Let us assume that  $H \vdash a \sim c$  and  $H \vdash b \sim d$  are true and let us prove that  $H \vdash a b \sim c d$  is true. Let  $H$  be  $x_1 : T_1, \dots, x_n : T_n$ . Let  $t_1, \dots, t_n, t'_1, \dots, t'_n$  be closed terms. We assume that:

$$x_1 : T_1, \dots, x_n : T_n \text{ @ } t_{(n)}$$

and

$$\forall i < n. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}]$$

We have to prove:

$$(a b \sim c d)[t_{(n)}/x_{(n)}] = (a b \sim c d)[t'_{(n)}/x_{(n)}]$$

and

$$\mathbf{Ax} = \mathbf{Ax} \in (a b \sim c d)[t_{(n)}/x_{(n)}]$$

Using our hypotheses, we obtain that:

$$\begin{aligned} (a \sim c)[t_{(n)}/x_{(n)}] &= (a \sim c)[t'_{(n)}/x_{(n)}] \\ (b \sim d)[t_{(n)}/x_{(n)}] &= (b \sim d)[t'_{(n)}/x_{(n)}] \\ \mathbf{Ax}=\mathbf{Ax} \in (a \sim c)[t_{(n)}/x_{(n)}] \\ \mathbf{Ax}=\mathbf{Ax} \in (b \sim d)[t_{(n)}/x_{(n)}] \end{aligned}$$

We conclude using the fact that  $\sim$  is a congruence.  $\square$

**Lemma 23.** *[squal-lam-D] is valid*

*Proof.* Let us assume that  $H, x : \mathbf{Base} \vdash a \sim b$  is true and let us prove that  $H \vdash \lambda x.a \sim \lambda x.b$  is true. Let  $H$  be  $x_1 : T_1, \dots, x_n : T_n$ . Let  $t_1, \dots, t_n, t'_1, \dots, t'_n$  be closed terms. We assume that:

$$x_1 : T_1, \dots, x_n : T_n @ t_{(n)}$$

and

$$\forall i < n. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}]$$

We have to prove:

$$(\lambda x.a \sim \lambda x.b)[t_{(n)}/x_{(n)}] = (\lambda x.a \sim \lambda x.b)[t'_{(n)}/x_{(n)}]$$

and

$$\mathbf{Ax}=\mathbf{Ax} \in (\lambda x.a \sim \lambda x.b)[t_{(n)}/x_{(n)}]$$

Using our first hypothesis, we obtain that:

$$\begin{aligned} \forall u. u \in \mathbf{Base} \\ \Rightarrow \left( (a \sim b)[t_{(n)}, u/x_{(n)}, x] = (a \sim b)[t'_{(n)}, u/x_{(n)}, x] \right) \\ \wedge \mathbf{Ax}=\mathbf{Ax} \in (a \sim b)[t_{(n)}, u/x_{(n)}, x] \end{aligned}$$

We conclude by definition of  $\sim$  [24].  $\square$

Note that other similar rules exist for the other operators of the language.

**Lemma 24.** *[squal-sql] is valid*

*Proof.* Let us assume that  $H \vdash a \leq b$  and  $H \vdash b \leq a$  are true and let us prove that  $H \vdash a \sim b$  is true. Let  $H$  be  $x_1 : T_1, \dots, x_n : T_n$ . Let  $t_1, \dots, t_n, t'_1, \dots, t'_n$  be closed terms. We assume that:

$$x_1 : T_1, \dots, x_n : T_n @ t_{(n)}$$

and

$$\forall i < n. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}]$$

We have to prove:

$$(a \sim b)[t_{(n)}/x_{(n)}] = (a \sim b)[t'_{(n)}/x_{(n)}]$$

and

$$\mathbf{Ax}=\mathbf{Ax}\in(a \sim b)[t_{(n)}/x_{(n)}]$$

Using our hypotheses, we obtain that:

$$(a \leq b)[t_{(n)}/x_{(n)}]=(a \leq b)[t'_{(n)}/x_{(n)}]$$

$$(b \leq a)[t_{(n)}/x_{(n)}]=(b \leq a)[t'_{(n)}/x_{(n)}]$$

$$\mathbf{Ax}=\mathbf{Ax}\in(a \leq b)[t_{(n)}/x_{(n)}]$$

$$\mathbf{Ax}=\mathbf{Ax}\in(b \leq a)[t_{(n)}/x_{(n)}]$$

We conclude by definition of  $\sim$ .  $\square$

*Remark 25 (Guaspari's trick).* Some of our rules require terms to be in **Base**. However, thanks to David Guaspari's trick, when proving simulations and bisimulations we can turn parameters of type **Top** into parameters of type **Base**. This trick mainly uses `[sqle-refl]` and `[squal-refl]`.

Let us assume that we trying to prove  $H, x : \mathbf{Top}, J \vdash a \sim b$ . By computation, we can turn that sequent into  $H, x : \mathbf{Top}, J \vdash (\lambda x.a) x \sim (\lambda x.b) x$ . By `[squal-app-D]`, it is enough to prove  $H, x : \mathbf{Top}, J \vdash \lambda x.a \sim \lambda x.b$  and  $H, x : \mathbf{Top}, J \vdash x \sim x$ . The second sequent it true by `[squal-refl]`. We prove the first sequent using `[squal-lam-D]`. It then remain to prove:  $H, x : \mathbf{Top}, J, y : \mathbf{Base} \vdash a[y/x] \sim b[y/x]$

**Lemma 26.** *[ispair-member] is valid*

*Proof.* Let us assume that  $H \vdash \mathbf{halts}(a)$ ,  $H \vdash b \in T$ ,  $H \vdash c \in T$ , and  $H \vdash a \in \mathbf{Base}$  are true. Let us now prove that  $H \vdash \mathbf{ispair}(a, b, c) \in T$  is true. Let  $H$  be  $x_1 : T_1, \dots, x_n : T_n$ . We have to prove that:

$$\forall t_{(n)}. x_1 : T_1, \dots, x_n : T_n \vdash \mathbf{ispair}(a, b, c) \in T @ t_{(n)}$$

i.e. given closed terms  $t_1, \dots, t_n, t'_1, \dots, t'_n$ , and assuming both  $x_1 : T_1, \dots, x_n : T_n @ t_{(n)}$  and  $\forall i < n. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}]$  then we have to prove:

$$(\mathbf{ispair}(a, b, c) \in T)[t_{(n)}/x_{(n)}]=(\mathbf{ispair}(a, b, c) \in T)[t'_{(n)}/x_{(n)}] \quad (12)$$

and

$$\mathbf{Ax}=\mathbf{Ax}\in(\mathbf{ispair}(a, b, c) \in T)[t_{(n)}/x_{(n)}] \quad (13)$$

Using the assumptions above, we obtain:

$$\mathbf{halts}(a)[t_{(n)}/x_{(n)}]=\mathbf{halts}(a)[t'_{(n)}/x_{(n)}] \quad (14)$$

$$(b \in T)[t_{(n)}/x_{(n)}]=(b \in T)[t'_{(n)}/x_{(n)}] \quad (15)$$

$$(c \in T)[t_{(n)}/x_{(n)}]=(c \in T)[t'_{(n)}/x_{(n)}] \quad (16)$$

$$(a \in \mathbf{Base})[t_{(n)}/x_{(n)}]=(a \in \mathbf{Base})[t'_{(n)}/x_{(n)}] \quad (17)$$



$$\mathbf{Ax}=\mathbf{Ax}\in\mathbf{halts}(a)[t_{(n)}/x_{(n)}] \quad (18)$$

$$\mathbf{Ax}=\mathbf{Ax}\in(b \in T)[t_{(n)}/x_{(n)}] \quad (19)$$

$$\mathbf{Ax}=\mathbf{Ax}\in(c \in T)[t_{(n)}/x_{(n)}] \quad (20)$$

$$\mathbf{Ax}=\mathbf{Ax}\in(a \in \mathbf{Base})[t_{(n)}/x_{(n)}] \quad (21)$$

Using 17 we obtain that:

$$a[t_{(n)}/x_{(n)}]=a[t'_{(n)}/x_{(n)}]\in\mathbf{Base}$$

Therefore  $a[t_{(n)}/x_{(n)}]$  and  $a[t'_{(n)}/x_{(n)}]$  have the same convergence behavior and if  $a[t_{(n)}/x_{(n)}]$  converges then  $a[t_{(n)}/x_{(n)}] \sim a[t'_{(n)}/x_{(n)}]$ . Using 14 and 18 we obtain that both:  $\mathbf{halts}(a[t_{(n)}/x_{(n)}])$  and  $\mathbf{halts}(a[t'_{(n)}/x_{(n)}])$ . It is then decidable whether this  $a[t_{(n)}/x_{(n)}]$  computes to a pair:

- If  $a[t_{(n)}/x_{(n)}]$  computes to a pair then  $a[t'_{(n)}/x_{(n)}]$  also computes to a pair because  $a[t_{(n)}/x_{(n)}] \sim a[t'_{(n)}/x_{(n)}]$ . Therefore  $\mathbf{ispair}(a, b, c)[t_{(n)}/x_{(n)}]$  computes to  $b[t_{(n)}/x_{(n)}]$  and  $\mathbf{ispair}(a, b, c)[t'_{(n)}/x_{(n)}]$  computes to  $b[t'_{(n)}/x_{(n)}]$ . We obtain that  $(b \in T)[t_{(n)}/x_{(n)}]=\mathbf{ispair}(a, b, c) \in T[t_{(n)}/x_{(n)}]$  and  $(b \in T)[t'_{(n)}/x_{(n)}]=\mathbf{ispair}(a, b, c) \in T[t'_{(n)}/x_{(n)}]$  and we conclude using 15 and 19.
- If  $a[t_{(n)}/x_{(n)}]$  does not compute to a pair then  $a[t'_{(n)}/x_{(n)}]$  does not compute to a pair because  $a[t_{(n)}/x_{(n)}] \sim a[t'_{(n)}/x_{(n)}]$ . Therefore  $\mathbf{ispair}(a, b, c)[t_{(n)}/x_{(n)}]$  computes to  $c[t_{(n)}/x_{(n)}]$  and  $\mathbf{ispair}(a, b, c)[t'_{(n)}/x_{(n)}]$  computes to  $c[t'_{(n)}/x_{(n)}]$ . We obtain that  $(c \in T)[t_{(n)}/x_{(n)}]=\mathbf{ispair}(a, b, c) \in T[t_{(n)}/x_{(n)}]$  and  $(c \in T)[t'_{(n)}/x_{(n)}]=\mathbf{ispair}(a, b, c) \in T[t'_{(n)}/x_{(n)}]$  and we conclude using 16 and 20.

□

The proofs that `[isaxiom-member]`, `[isinl-member]`, `[isinr-member]`, `[isint-member]`, and `[islamba-member]` are valid are similar to the one above.

**Lemma 27.** *[ispair] is valid*

*Proof.* Let us assume that  $H \vdash \mathbf{ispair}(t, \mathbf{inl}(a), \mathbf{inr}(b)) \sim \mathbf{inl}(a)$  is true. Let us now prove that  $H \vdash t \in \mathbf{Top} \times \mathbf{Top}$  is true. Let  $H$  be  $x_1 : T_1, \dots, x_n : T_n$ . Let  $t_1, \dots, t_n, t'_1, \dots, t'_n$  be closed terms. Let us assume that  $x_1 : T_1, \dots, x_n : T_n @ t_{(n)}$  and  $\forall i < n. t_{i+1}=t'_{i+1} \in T_{i+1}[t_i/x_i]$  then we have to prove:

$$t[t_{(n)}/x_{(n)}] \in \mathbf{Top} \times \mathbf{Top}=t[t'_{(n)}/x_{(n)}] \in \mathbf{Top} \times \mathbf{Top} \quad (22)$$

and

$$\mathbf{Ax}=\mathbf{Ax}\in t[t_{(n)}/x_{(n)}] \in \mathbf{Top} \times \mathbf{Top} \quad (23)$$

Using our first assumption, we obtain:

$$\begin{aligned} &(\text{ispair}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inl}(a))[t_{(n)}/x_{(n)}] \\ &= (\text{ispair}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inl}(a))[t'_{(n)}/x_{(n)}] \end{aligned} \quad (24)$$

$$\text{Ax}=\text{Ax}\in(\text{ispair}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inl}(a))[t_{(n)}/x_{(n)}] \quad (25)$$

Because 25 is true,  $t[t_{(n)}/x_{(n)}]$  must compute to a pair, i.e., 23 is true. Because 24 is true, and because  $(\text{ispair}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inl}(a))[t_{(n)}/x_{(n)}]$  is true, then  $(\text{ispair}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inl}(a))[t'_{(n)}/x_{(n)}]$  must also be true, i.e.,  $t[t'_{(n)}/x_{(n)}]$  must compute to a pair. Knowing that both  $t[t_{(n)}/x_{(n)}]$  and  $t[t'_{(n)}/x_{(n)}]$  compute to pairs is enough to prove 22.  $\square$

The proofs that [isaxiom], [isinl], [isinr], [isint], and [islambda] are valid are similar to the one above.

**Lemma 28.** *[not-ispair] is valid*

*Proof.* Let us assume that  $H \vdash \text{ispair}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inr}(b)$  is true. Let us now prove that  $H \vdash \text{ispair}(t, c, d) \sim d$  is true. Let  $H$  be  $x_1 : T_1, \dots, x_n : T_n$ . Let  $t_1, \dots, t_n, t'_1, \dots, t'_n$  be closed terms. Let us assume that  $x_1 : T_1, \dots, x_n : T_n @ t_{(n)}$  and  $\forall i < n. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}]$  then we have to prove:

$$(\text{ispair}(t, c, d) \sim d)[t_{(n)}/x_{(n)}] = (\text{ispair}(t, c, d) \sim d)[t'_{(n)}/x_{(n)}] \quad (26)$$

and

$$\text{Ax}=\text{Ax}\in(\text{ispair}(t, c, d) \sim d)[t_{(n)}/x_{(n)}] \quad (27)$$

Using our first assumption, we obtain:

$$\begin{aligned} &(\text{ispair}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inr}(b))[t_{(n)}/x_{(n)}] \\ &= (\text{ispair}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inr}(b))[t'_{(n)}/x_{(n)}] \end{aligned} \quad (28)$$

$$\text{Ax}=\text{Ax}\in(\text{ispair}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inr}(b))[t_{(n)}/x_{(n)}] \quad (29)$$

Because 29 is true, we know that  $t[t_{(n)}/x_{(n)}]$  must compute to a value that is not a pair. Therefore 27 is true. Because 29 and  $(\text{ispair}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inr}(b))[t_{(n)}/x_{(n)}]$  are true, then  $(\text{ispair}(t, \text{inl}(a), \text{inr}(b)) \sim \text{inr}(b))[t'_{(n)}/x_{(n)}]$  must also be true, i.e.,  $t[t'_{(n)}/x_{(n)}]$  must compute to a value that is not a pair. Therefore  $\text{Ax}=\text{Ax}\in(\text{ispair}(t, c, d) \sim d)[t'_{(n)}/x_{(n)}]$  is true and we get that 26 is true.  $\square$

The proofs that [not-isaxiom], [not-isinl], [not-isinr], [not-isint], and [not-islambda] are valid are similar to the one above.

**Lemma 29.** *[convergence] is valid*

*Proof.* Let us assume that  $H, y : \text{halts}(a) \vdash a \leq b$  and  $H \vdash \text{halts}(a) \in \mathbb{P}$  are true. Let us now prove that  $H \vdash a \leq b$  is true. Let  $H$  be  $x_1 : T_1, \dots, x_n : T_n$ . Let  $t_1, \dots, t_n, t'_1, \dots, t'_n$  be closed terms. Let us assume that  $x_1 : T_1, \dots, x_n : T_n @ t_{(n)}$  and  $\forall i < n. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}]$  then we have to prove:

$$(a \leq b)[t_{(n)}/x_{(n)}] = (a \leq b)[t'_{(n)}/x_{(n)}] \quad (30)$$

and

$$\text{Ax} = \text{Ax} \in (a \leq b)[t_{(n)}/x_{(n)}] \quad (31)$$

From our first hypothesis that  $H, y : \text{halts}(a) \vdash a \leq b$  is true, we obtain

$$\begin{aligned} & \left( \begin{array}{l} \text{Ax} \in \text{halts}(a)[t_{(n)}/x_{(n)}] \\ \wedge \forall t'_{(n)}. \forall i < n. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}] \\ \Rightarrow \text{halts}(a)[t_{(n)}/x_{(n)}] = \text{halts}(a)[t'_{(n)}/x_{(n)}] \end{array} \right) \quad (32) \\ \Rightarrow & \left( \begin{array}{l} (a \leq b)[t_{(n)}/x_{(n)}] = (a \leq b)[t'_{(n)}/x_{(n)}] \\ \wedge \text{Ax} \in (a \leq b)[t_{(n)}/x_{(n)}] \end{array} \right) \end{aligned}$$

From our second hypothesis that  $H \vdash \text{halts}(a) \in \mathbb{P}$  is true, we obtain

$$\begin{aligned} & \forall t'_{(n)}. \forall i < n. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}] \quad (33) \\ \Rightarrow & \left( \begin{array}{l} (\text{halts}(a) \in \mathbb{P})[t_{(n)}/x_{(n)}] = (\text{halts}(a) \in \mathbb{P})[t'_{(n)}/x_{(n)}] \\ \wedge \text{Ax} \in (\text{halts}(a) \in \mathbb{P})[t_{(n)}/x_{(n)}] \end{array} \right) \end{aligned}$$

Let us prove 31, i.e., we have to prove that  $(a \leq b)[t_{(n)}/x_{(n)}]$  is true. By definition of  $\leq$  we can assume that  $a[t_{(n)}/x_{(n)}]$  has a value. Therefore the first hypothesis of 32 is true. Its second hypothesis is also true because it is implied by 33. Therefore, we obtain 31 which is the second conclusion of 32.

Let us now prove 30. Because 31 is true, it remains to prove that  $(a \leq b)[t'_{(n)}/x_{(n)}]$  is also true. By definition of  $\leq$  we can assume that  $a[t'_{(n)}/x_{(n)}]$  has a value. Using 33, we obtain  $(\text{halts}(a) \in \mathbb{P})[t_{(n)}/x_{(n)}] = (\text{halts}(a) \in \mathbb{P})[t'_{(n)}/x_{(n)}]$  which implies that  $\text{halts}(a)[t_{(n)}/x_{(n)}] = \text{halts}(a)[t'_{(n)}/x_{(n)}]$  is true. By definition of  $\text{halts}$ , and given that  $a[t'_{(n)}/x_{(n)}]$  has a value, we get that  $\text{halts}(a)[t_{(n)}/x_{(n)}]$  is also true. Finally, we conclude as above.  $\square$

**Lemma 30.** *[halt-spread] is valid*

*Proof.* Let us assume that  $H \vdash \text{halts}(\text{let } x, y = p \text{ in } F)$  is true. Let us now prove that  $H \vdash p \in \text{Top} \times \text{Top}$  is true. Let  $H$  be  $x_1 : T_1, \dots, x_n : T_n$ . Let  $t_1, \dots, t_n, t'_1, \dots, t'_n$  be closed terms. Let us assume that  $x_1 : T_1, \dots, x_n : T_n @ t_{(n)}$  and  $\forall i < n. t_{i+1} = t'_{i+1} \in T_{i+1}[t_{(i)}/x_{(i)}]$  then we have to prove:

$$(p \in \text{Top} \times \text{Top})[t_{(n)}/x_{(n)}] = (p \in \text{Top} \times \text{Top})[t'_{(n)}/x_{(n)}] \quad (34)$$

and

$$\text{Ax} = \text{Ax} \in (p \in \text{Top} \times \text{Top})[t_{(n)}/x_{(n)}] \quad (35)$$

Using our first assumption, we obtain:

$$\begin{aligned} & (\text{halts}(\text{let } x, y = p \text{ in } F))[t_{(n)}/x_{(n)}] \quad (36) \\ & = (\text{halts}(\text{let } x, y = p \text{ in } F))[t'_{(n)}/x_{(n)}] \end{aligned}$$

$$\text{Ax}=\text{Ax}\in(\text{halts}(\text{let } x, y = p \text{ in } F))[t_{(n)}/x_{(n)}] \quad (37)$$

Because 37 is true and by definition of `halts`, we know that  $p[t_{(n)}/x_{(n)}]$  must compute to a pair. We obtain that 35 is true. Because 36 is true, and because  $(\text{halts}(\text{let } x, y = p \text{ in } F))[t_{(n)}/x_{(n)}]$  is true, then  $(\text{halts}(\text{let } x, y = p \text{ in } F))[t'_{(n)}/x_{(n)}]$  must also be true, i.e.,  $p[t'_{(n)}/x_{(n)}]$  must compute to a pair. Knowing that both  $p[t_{(n)}/x_{(n)}]$  and  $p[t'_{(n)}/x_{(n)}]$  compute to pairs is enough to prove 34.  $\square$

The proofs that the other rules presented in Fig. 8 are valid are similar to the one above.

## F Nuprl Proofs

This section contains our Nuprl proofs of the lemmas presented in this paper. In these Nuprl statements and proofs,  $(t)\downarrow$  is our way of displaying `halts(t)`. As mentioned above the tactic `SqReasoning` implements part of the reasoning used in this paper to prove bisimulations. The tactics `HVimplies` and `HVimpliesRec` instantiate the lemmas such as Lemma 2. The tactic `OneFixpointLeast` calls on the `[least-upper-bound]` rule. This section does not include the proofs of Lemma 10, Lemma 11, and lemma 12 because, given our formal definition of the `process` function and given the format of this paper, they would not be readable. We are planing on exporting these lemmas and their proofs to html.

### F.1 Lemma 1

---

```

 $\vdash \forall [t:\text{Base}]. \text{ispair}(t) \in \mathbb{B} \text{ supposing } (t)\downarrow$ 
|
BY (Auto THEN IspairMember THEN Auto)

```

---

### F.2 Lemma 2

---

```

 $\vdash \forall t, a, b:\text{Base}. ((t)\downarrow \Rightarrow ((t \sim \langle \text{fst}(t), \text{snd}(t) \rangle$ 
 $\quad \vee (\text{if } t \text{ is a pair then } a \text{ otherwise } b \sim b)))$ 
|
BY (Auto
| THEN (FLemma 'ispair-bool-if-has-value' [-1] THENA Auto)
| THEN (InstLemma 'bool_cases_sqeual' [↑ispair(t)]· THENA Auto)
| THEN D (-1))
|\
| 1. t: Base
| 2. a: Base
| 3. b: Base
| 4. (t)↓
| 5. ispair(t) ∈ ℬ
| 6. ispair(t) ∼ tt
| ⊢ (t ∼ ⟨fst(t), snd(t)⟩) ∨ (if t is a pair then a otherwise b ∼ b)
| |

```

---

```

1 BY ((OrLeft THENA Auto) THEN IsPair (-1) THEN AutoPairEta [1] 0)
  \
  1. t: Base
  2. a: Base
  3. b: Base
  4. (t)↓
  5. ispair(t) ∈ ℬ
  6. ispair(t) ∼ ff
  ⊢ (t ∼ <fst(t), snd(t)>) ∨ (if t is a pair then a otherwise b ∼ b)
  |
  BY ((OrRight THENA Auto)
      THEN NotIsPairAux 「.」 「.」.
      THEN Folds ‘btrue bfalse‘ 0
      THEN Auto)

```

---

### F.3 Lemma 4

```

⊢ ∀[p,F,G:Top].
|   (let c,d = let a,b = p
|       in F[a;b]
|   in G[c;d] ∼ let a,b = p
|       in let c,d = F[a;b]
|       in G[c;d])
|
BY (SqReasoning
    THEN (Assert 「(let a,b = p in F[a;b])↓」. THENA ProveHasValue)
    THEN HasValueD (-1)
    THEN AutoPairEta [2;1] (-4)
    THEN Repeat (AutoPairEta [2;1] 0))

```

---

### F.4 Lemma 5

```

⊢ ∀[a,b,c,d,e:Top].
|   (if (if a is a pair then b otherwise c) is a pair then d otherwise e
|   ∼ if a is a pair then (if b is a pair then d otherwise e)
|   otherwise (if c is a pair then d otherwise e))
|
BY (SqReasoning
    THEN HVimpliesRec 0 [1]
    THEN HVimpliesRec 0 [2]
    THEN HypSubstAll (-2)
    THEN Try (Complete ((HVimpliesRec (-2) [1] THEN HypSubstAll (-3))))
    THEN HVimpliesRec (-2) [1]
    THEN HVimpliesRec (-1) [1]
    THEN HypSubstAll (-2)
    THEN HypSubstAll (-4)
    THEN Try (Complete ((HVimpliesRec (-1) [1;1] THEN HypSubstAll (-2))))
    THEN Try (Complete ((HVimpliesRec (-1) [1] THEN HypSubstAll (-2))))

```

---

**F.5 Lemma 6**


---

```

⊢ ∀[p,F:Top]. (let a,b = p in F[p] ~ let a,b = p in F[<a, b>])
|
BY SqReasoning

```

---

**F.6 Lemma 7**


---

```

⊢ ∀[F:Top]. (let a,b = bottom() in F[a;b] ~ bottom())
|
BY (SqReasoning THEN Assert 「(bottom())↓」. THEN BotDiv THEN ProveHasValue)

```

---

**F.7 Lemma 8**


---

```

⊢ ∀[t:Top]. (t @ [] ≤ t)
|
BY (Auto
    THEN RepUR “append list_ind nil it cons“ 0
    THEN OneFixpointLeast
    THEN MoveToConcl (-2)
    THEN NatInd (-1)
    THEN (UnivCD THENA Auto)
    THEN Try (Complete ((Reduce 0 THEN Strictness THEN Auto)))
    THEN (RWO "fun_exp_unroll_1" 0 THENA Auto)
    THEN Reduce 0
    THEN SqReasoning
    THEN Repeat (HVimplies 0 [1]))

```

---

**F.8 Lemma 9**

To prove that lemma, we follow the proof depicted in Sec. A instead of following the proof depicted in Sec. 4.7.

---

```

⊢ ∀[f,b:Top]. (map(f;b) @ [] ~ map(f;b))
|
BY (SqReasoning
    THEN Try (Complete ((BLemma ‘append-nil-sqle‘ THEN Auto)))
    THEN (BLemma ‘sqle-append-nil-if-has-value3‘ THENA Auto)
    THEN BLemma ‘is-list-if-has-value-rec-map‘
    THEN Auto)

```

---