Equality In Lazy Computation Systems[*]

Douglas J. Howe
Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract

In this paper we introduce a general class of lazy computation systems and define a natural program equivalence for them. We prove that if an *extensionality* condition holds of each of the operators of a computation system, then the equivalence relation is a congruence, so that the usual kinds of equality reasoning are valid for it. This condition is a simple syntactic one, and is easy to verify for the various lazy computation systems we have considered so far. We also give conditions under which the equivalence coincides with observational congruence. These results have some important consequences for type theories like those of Martin-Löf and Nuprl.

1 Introduction

In a lazy programming language, evaluation terminates when the outermost data constructor of the result becomes known. Thus a pair $\langle a, b\rangle$ is considered as fully evaluated regardless of what a and b are. This paper addresses the problem of reasoning about program equivalence in lazy languages. We define a broad class of such languages, and define a natural notion of program equivalence for them. In order for normal equality reasoning to be valid for this equivalence, it must be possible to replace a program fragment by an equivalent one in any context. Our main result is a reduction of this property to a condition on each of the individual language constructs. The condition seems to be generally easy to verify; we have checked it for a variety of examples.

The original motivation for this work was to justify certain useful kinds of type-free inference for the type theory of the Nuprl proof development system [8, 3, 2]. The type theories of Nuprl and Martin-Löf [10] are based on an untyped computation system. A type system is constructed by selecting certain terms of the computation system to denote types and by specifying for each type what terms are its members and when two members of the type are to be considered equal. A major practical problem with these type theories has been the lack of inference rules that do not involve reasoning about types. For example, a proof of the equality of two types that differ only in the contraction of a β-redex would in general require finding a type for the redex and proving that the type was appropriate. More generally, although it seemed very likely that the judgements of the type system were preserved under the reductions that generate the evaluation relation of the computation system, no proof of this fact was known.

An attempt to prove that a binary relation R on closed terms is respected by the judgments of the type system reveals (see, for example, [2]) a sufficient condition: that R is a congruence (that is, R is respected by the operators of the term language) and $R \subset [R]$, where $a\ [R]\ b$ if, roughly, whenever evaluation of a terminates then so does the evaluation of b, and furthermore the resulting values have the same outermost form and the corresponding components are related by R. For example, in the case of the λ-calculus, $a\ [R]\ a'$ means that whenever a evaluates to an abstraction $\lambda x.\, b$, a' must evaluate to some abstraction $\lambda x'.\, b'$ such that $b[c/x]\ R\ b'[c/x']$ for all closed terms c. Since one of our goals is to justify as much type-free inference as possible, we desire the largest possible R satisfying these conditions. We take the largest R satisfying $R \subset [R]$ and then prove it is a congruence. We call this a *maximal congruence.*

We start by defining a *lazy computation system* or *lcs.* The class of lazy computation systems is a direct generalization of the computation system of Martin-Löf's type theory [10]. An lcs consists of a term language, where the operators of the language may have

[*] This research was supported in part by NSF grant CCR-8616552 and ONR grant N00014-88-K-0409

variable-binding structure, together with an evaluation relation which can be any relation on closed terms that is the identity function on those terms that are taken to be values. For an lcs $\mathcal{S}$ we define a preorder $\leq$ on the closed terms of $\mathcal{S}$ as the greatest fixed-point of $[\cdot]$. We then define an *extensionality* condition on the operators of $\mathcal{S}$ and show that if every operator is extensional (that is, $\mathcal{S}$ is an *extensional lcs*) then $\leq$ is a congruence. We have constructed various examples of extensional lcs. The examples include usual kinds of lazy functional computation, as well as nondeterministic and call-by-value computation. In each example, the verification of operator extensionality was simple.

A natural notion of program equivalence, at least for deterministic computation, is *observational congruence*, where two programs are equivalent if they have the same observable behaviour in all program contexts. For an lcs we take the basic observation to be termination of evaluation together with the identity of the outermost operator in the resulting value. We prove that in any deterministic lcs that satisfies a minimal computational adequacy condition, observational congruence is the same as the equivalence $\sim$ defined as $\leq \cap \geq$.

In an lcs which is the λ-calculus with some added constants and reductions, this equivalence of $\sim$ with observational congruence has become known as *observational extensionality* [6]. In this setting, $a \leq b$ if and only if for all sequences of closed terms $c_1, \ldots, c_n$, if evaluation of $ac_1 \ldots c_n$ terminates, then so does evaluation of $bc_1 \ldots c_n$ and the values have the same outermost form. It is easy to show that $\leq$ being a congruence is equivalent to observational extensionality. In his paper on the lazy λ-calculus [1], Abramsky proved observational extensionality for the pure λ-calculus and several simple extensions. His proofs involve a complicated excursion through domain theory and domain logic, and his method does not appear to generalize. In [6], Bloom proves observational extensionality for a class of extensions to LCF. His proof applies to a special class of languages; for example, they must be simply typed. In [5], Berry gives a proof which when appropriately specialized gives a simple argument from first principles that the pure lazy λ-calculus is observationally extensional. This proof can be modified to work in several simple extensions of the λ-calculus, but such modifications seem *ad hoc*.

In contrast, our results apply to a broad class of languages, and we reduce $\leq$ being a congruence to a condition on evaluation that appears to be very simple to verify in general. For example, the verification for each of the operators in Martin-Löf's type theory can be done in a few lines. In addition to easy verification, the form of our extensionality condition permits the inclusion of the condition in an *open-ended* account of computation in intuitionistic type theory. From an intuitionistic viewpoint, open-endedness, which requires that the definition of each operator and the definition of evaluation anticipate the introduction of further operators, is essential if type theory is to be a foundation for constructive mathematics. For a discussion of some of the issues related to type-theoretic open-endedness, see [3].

Our presentation assumes a basic familiarity with such syntactic notions of the λ-calculus as free and bound variable and substitution. We will use

$$b[a_1, \ldots a_n/x_1, \ldots, x_n]$$

to denote the simultaneous substitution in the term b of the terms a_1, ..., a_n for the variables x_1, ..., x_n. We will write $b[a_1, \ldots, a_n]$ when the variables x_1, ..., x_n can be inferred from context. In Section 2 we define our class of lazy computation systems. In Section 3 we prove our main result, that in an extensional lcs $\sim$ is a congruence. We also briefly discuss observational congruence. In the last section we discuss some applications.

2 Lazy Computation Systems

Define a *lazy computation language* to be a triple $\langle O, K, \alpha\rangle$ such that

- O is a set,
- $K \subset O$ and
- $\alpha \in O \rightarrow \{\, \langle k_1, \ldots, k_n\rangle \mid n, k_i \geq 0 \,\}$.

We call the members of O *operators*, and the members of K *canonical* operators. For $\tau \in O$, $\alpha(\tau)$ is the *arity* of τ and specifies the number and binding structure of the operator's possible arguments.

Let L be a lazy computation language $\langle O, K, \alpha\rangle$. To define the set $T(L)$ of *terms* over L we first inductively define sets $B_n(L)$ for $n \geq 0$ as follows. Fix an infinite set V of variables.

- $V \subset B_0(L)$.
- If $t \in B_0(L)$ and $x_1, \ldots, x_n \in V$ are distinct then $x_1, \ldots, x_n . t \in B_n(L)$.
- If $\alpha(\tau) = \langle k_1, \ldots, k_n\rangle$ and $t_i \in B_{k_i}(L)$ for $1 \leq i \leq n$ then $\tau(t_1, \ldots, t_n) \in B_0(L)$.

Let $T(L) = B_0(L)$. We give variable-binding structure to $T(L)$ by specifying that in a member $x_1, \ldots, x_n.\, b$ of $B_n(L)$ the free occurrences of $x_1, \ldots, x_n$ in b become bound. Define $T^0(L)$ to be the set of all closed terms (terms without free variables) in $T(L)$. The closed terms will be the programs of our computation systems. A *canonical* term is a closed term of the form $\tau(t_1, \ldots, t_n)$ where τ is a canonical operator. The canonical terms will be exactly the results of evaluating programs; thus a term will be considered to be fully evaluated exactly if its outermost operator is canonical. A *noncanonical* term is a closed term that is not canonical. We will often use an overline notation for a (possibly empty) sequence of operands; for example, we write $\tau(\overline{t})$ for $\tau(t_1, \ldots, t_n)$. We identify terms that are α-equal (the same up to renaming of bound variables).

A *lazy computation system*, or *lcs*, is a lazy computation language L together with binary relations $\xrightarrow{k}$ on $T^0(L)$ ($k \geq 0$) such that

- for all $k \geq 1$, if $a \xrightarrow{k} b$ then a is noncanonical and b is canonical, and
- $a \xrightarrow{0} b$ if and only if a is canonical and $a = b$.

Define $a \longrightarrow b$ if there is some $k \geq 0$ such that $a \xrightarrow{k} b$. We call $\longrightarrow$ the *evaluation* relation of the lcs; b is a *value* of a if $a \longrightarrow b$. Define $a \xrightarrow{<k} b$ if there exists a $j < k$ such that $a \xrightarrow{j} b$. The evaluation relation is *deterministic* if for every closed a, b and c, $a \longrightarrow b$ and $a \longrightarrow c$ imply $b = c$. Note that since we are identifying α-equal terms, evaluation must respect α-equality.

For example, the usual untyped λ-calculus can be viewed as an lcs. Take $O = \{\lambda, ap\}$ and $K = \{\lambda\}$. Define α by $\alpha(\lambda) = 1$ and $\alpha(ap) = \langle 0, 0\rangle$. Write $\lambda x.\, b$ for $\lambda(x.\, b)$, and $a(b)$ for $ap(a, b)$. Define $a \xrightarrow{0} a'$ if $a = a' = \lambda x.\, b$ for some b. For $k \geq 1$, define $a \xrightarrow{k} a'$ if a is of the form $(\lambda x.\, b)(a_1) \ldots (a_n)$ and $b[a_1/x](a_2) \ldots (a_n) \xrightarrow{k-1} a'$. Call this lcs Λ.

To simplify the presentation of our work we use a few notational conventions. Let R be a binary relation on $T(L)$. If $t = x_1, \ldots, x_n.\, b$ and $t' = x'_1, \ldots, x'_n.\, b'$ are members of $B_n(L)$, then when we write $t \; R \; t'$ we mean that there exist distinct variables $z_1, \ldots, z_n$ not free in t or t' such that

$$b[z_1, \ldots, z_n/x_1, \ldots, x_n] \; R \; b'[z_1, \ldots, z_n/x'_1, \ldots, x'_n].$$

If $\tau(t_1, \ldots, t_n)$ and $\tau(t'_1, \ldots, t'_n)$ are members of $T(L)$, then when we write $\overline{t} \; R \; \overline{t'}$ we mean that $t_i \; R \; t'_i$ for each i, $1 \leq i \leq n$. Finally, if R is a binary relation on $T^0(L)$, then $a \; R \; b$ for $a, b \in T(L)$ means that $\sigma(a) \; R \; \sigma(b)$ for every substitution σ such that $\sigma(a)$ and $\sigma(b)$ are closed.

We could have simplified the syntax of an lcs by taking lambda-abstraction as the only binding construct. However, this would not appreciably simplify what follows, and it would be more difficult to directly apply our results to the type theories we are interested in.

The decomposition of evaluation in an lcs into a set of relations indexed by N is somewhat arbitrary. With a few trivial modifications to our definitions and proofs, N can be replaced by an arbitrary well-founded relation.

3 Equality

For this section we fix an arbitrary lcs $\mathcal{S}$ with language $L = \langle O, K, \alpha\rangle$ and evaluation relations $\{\xrightarrow{k}\}$.

Definition 1 *Let R be a binary relation on $T^0(L)$. For closed a and a', define $a \; [R] \; a'$ if whenever $a \longrightarrow \theta(\overline{t})$ for some canonical $\theta(\overline{t})$ there is a closed $\theta(\overline{t'})$ such that $a' \longrightarrow \theta(\overline{t'})$ and $\overline{t} \; R \; \overline{t'}$.*

Definition 2 *Define $\leq$ as the largest relation R on closed terms such that $R \subset [R]$.*

This exists since $[\cdot]$ is monotonic, and we have $\leq = [\leq]$. (In the case of deterministic evaluation, we could define $\leq$ as $\bigcap_{n \geq 0} \leq_n$, where $\leq_0$ relates all closed terms and $\leq_{n+1}$ is $[\leq_n]$.) This relation, hence also its extension to $T(L)$, is reflexive and transitive. The equivalence relation we are interested in is defined as follows.

Definition 3 *$a \sim b$ if $a \leq b$ and $b \leq a$.*

3.1 Extensionality

We want $\sim$ to be a congruence with respect to the operators of L. In other words, we want $\tau(\overline{t}) \leq \tau(\overline{t'})$ whenever $\overline{t} \leq \overline{t'}$. We define a relation that has this property by definition and that contains $\leq$. We will then show that this is the same as $\leq$ when each operator of $\mathcal{S}$ satisfies an *extensionality* condition.

Definition 4 *Define $a \leq^* b$, for $a, b \in T(L)$, by induction on the construction of a.*

- *For a variable x, $x \leq^* b$ if $x \leq b$.*
- *$\tau(\overline{t}) \leq^* b$ if there is a $\tau(\overline{t'})$ such that $\overline{t} \leq^* \overline{t'}$ and $\tau(\overline{t'}) \leq b$.*

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 $\leq$. The following are immediate consequences of Definition 4.

- $\tau(\overline{t}) \leq^* \tau(\overline{t'})$ if $\overline{t} \leq^* \overline{t'}$.
- $t \leq^* t$ for all t.
- If $a \leq^* b$ and $b \leq c$ then $a \leq^* c$.
- If $a \leq b$ then $a \leq^* b$.
- For any substitution σ of variables for variables, if $t \leq^* t'$ then $\sigma(t) \leq^* \sigma(t')$.

In what follows, these facts will be used without explicit reference.

Lemma 1 *If $a \leq^* a'$ and $b \leq^* b'$ then $b[a/x] \leq^* b'[a'/x]$.*

Proof. The proof is by induction on the size of b. In the case where b is the variable x, we have $x \leq b'$, and so $a' \leq b'[a'/x]$ by the definition of $\leq$ on open terms. Since $a \leq^* a'$, $a \leq^* b'[a'/x]$. In the case that b is a variable $y \neq x$, since $y \leq b'$ we have $y \leq b'[a'/x]$. For the induction step, we have $b = \tau(\overline{t})$, $\overline{t} \leq^* \overline{t'}$ and $\tau(\overline{t'}) \leq b'$. $\tau(\overline{t})[a/x] \leq^* \tau(\overline{t'})[a'/x]$ follows from the induction hypothesis. Also, $\tau(\overline{t'})[a'/x] \leq b'[a'/x]$, so $\tau(\overline{t})[a/x] \leq^* b'[a'/x]$. □

Lemma 2 *If $\theta(\overline{t})$ and b are closed terms such that $\theta(\overline{t})$ is canonical and $\theta(\overline{t}) \leq^* b$, then there is a closed $\theta(\overline{t'})$ such that $b \rightarrow \theta(\overline{t'})$ and $\overline{t} \leq^* \overline{t'}$.*

Proof. The definition of $\leq^*$ gives a term $\theta(\overline{t''})$ such that $\overline{t} \leq^* \overline{t''}$ and $\theta(\overline{t''}) \leq b$. By Lemma 1 and the definition of $\leq$ on open terms, we may assume that $\theta(\overline{t''})$ is closed. Since $\leq = [\leq]$, there is a closed $\theta(\overline{t'})$ such that $b \rightarrow \theta(\overline{t'})$ and $\overline{t''} \leq \overline{t'}$. Since $\overline{t} \leq^* \overline{t''}$, $\overline{t} \leq^* \overline{t'}$. □

We have defined $\leq^*$ to be a minimal congruence refined by $\leq$. Since $\leq$ is defined as the maximal fixed-point of $[\cdot]$, to show $\leq^*$ is in fact the same as $\leq$ it suffices to show that $\leq^*$ respects evaluation. To establish this condition it suffices to show that each operator inductively preserves it: given a pair of operand sequences $\overline{t}$ and $\overline{t'}$ for an operator τ, if $\overline{t} \leq^* \overline{t'}$ and the condition is inductively assumed to be true, then the values of $\tau(\overline{t})$ and $\tau(\overline{t'})$ are related by $\leq^*$. We call this property of an operator "operator extensionality" because proving it amounts to showing that two applications of an operator are $\sim$ if the operands are, so that the value of an application depends only on the values of the closed instances of the operands.

Definition 5 *An operator τ is* extensional *if for any closed terms $\tau(\overline{t})$, $\tau(\overline{t'})$ and a, and for any $k \geq 0$, if*

1. *$\tau(\overline{t}) \xrightarrow{k} a$,*
2. *$\overline{t} \leq^* \overline{t'}$ and*
3. *for every closed u, u' and v, if $u \xrightarrow{\leq k} u'$ and $u \leq^* v$ then $u' \leq^* v$,*

then $a \leq^* \tau(\overline{t'})$.

An lcs is extensional if all of its operators are. Note that canonical operators are always extensional since if τ is canonical and $\tau(\overline{t}) \xrightarrow{k} a$ then $k = 0$ and $\tau(\overline{t}) = a$.

Definition 5 was simply abstracted from the proof of Theorem 1, from an argument by induction on k that if $a \leq^* b$ and $a \xrightarrow{k} a'$ then $a' \leq^* b$. This argument shows that $\leq^*$ respects evaluation, since if $a \rightarrow a'$ and $a \leq^* b$, then $a' \leq^* b$ exactly if there is a b' such that $b \rightarrow b'$ and $a' \leq^* b'$ (this is an easy consequence of Lemma 2 and the fact that $b \rightarrow b'$ implies $b' \leq b$).

We can now state and prove the main result.

Theorem 1 *If $\mathcal{S}$ is extensional then for all closed a and b, $a \leq^* b$ implies $a \leq b$.*

Proof. We first show by induction on k that for every closed a, a' and b, if $a \xrightarrow{k} a'$ and $a \leq^* b$ then $a' \leq^* b$. If $k = 0$ then $a = a'$. Suppose that $k > 0$, and write a as $\tau(\overline{t})$. By definition of $\leq^*$ there exists $\overline{t'}$ such that $\overline{t} \leq^* \overline{t'}$ and $\tau(\overline{t'}) \leq b$. By Lemma 1, we may take $\tau(\overline{t'})$ to be closed. Since τ is extensional, and since our induction hypothesis is just *3* in Definition 5, $a' \leq^* \tau(\overline{t'})$. Since $\tau(\overline{t'}) \leq b$, we have $a' \leq^* b$.

Let R be the restriction of $\leq^*$ to closed terms. To prove the theorem it suffices to show that $R \subset [R]$. Suppose, then, that $a \; R \; b$ and $a \rightarrow \theta(\overline{t})$. By what we just proved, $\theta(\overline{t}) \leq^* b$. By Lemma 2, $b \rightarrow \theta(\overline{t'})$ for some $\overline{t'}$ such that $\overline{t} \leq^* \overline{t'}$. By Lemma 1, $\overline{t} \; R \; \overline{t'}$. □

The following theorem is now easy to prove.

Theorem 2 *Suppose that $\mathcal{S}$ is extensional. Then*

1. *for all $a, b \in T(L)$, $a \leq b$ if and only if $a \leq^* b$, and*
2. *if $\tau(\overline{t}), \tau(\overline{t'}) \in T(L)$ and if $\overline{t} \sim \overline{t'}$, then $\tau(\overline{t}) \sim \tau(\overline{t'})$.*

A converse property also holds. In any lcs where $\tau(\overline{t}) \leq \tau(\overline{t'})$ whenever $\overline{t} \leq \overline{t'}$, $\leq$ and $\leq^*$ are the same and so every operator is extensional.

As an example, we verify that the lcs Λ (defined in Section 2) is extensional. We only need to check ap. Suppose, then, that $f(a)$ and $f'(a')$ are closed, $f \leq^* f'$, $a \leq^* a'$, $f(a) \xrightarrow{k} c$, and part *3* of Definition 5 holds. There is a b such that $f \xrightarrow{<k} \lambda x.\, b$, and so $\lambda x.\, b \leq^* f'$. By Lemma 2, $f' \rightarrow \lambda x.\, b'$ for some b' such that $b \leq^* b'$. By Lemma 1, $b[a/x] \leq^* b'[a'/x]$. Since $b[a/x] \xrightarrow{<k} c$, $c \leq^* b'[a'/x] \leq f'(a')$, so $c \leq^* f'(a')$.

An argument similar to the above shows that Λ is extensional under call-by-value evaluation of applications. Nondeterministic lcs's can be also be extensional. For example, suppose the operator amb is evaluated as follows: if $a \rightarrow c$ or $b \rightarrow c$ then $amb(a, b) \rightarrow c$. We show amb is extensional. Suppose that $a \leq^* a'$, $b \leq^* b'$ and $amb(a, b) \xrightarrow{k} c$. Either $a \xrightarrow{<k} c$ or $b \xrightarrow{<k} c$. In the first case $c \leq^* a' \leq amb(a', b')$ and so $c \leq^* amb(a', b')$. The other case is similar.

3.2 Observational Congruence

Define $\leq_1$ to be $[true]$. For $a, b \in T(L)$, define $a \leq_C b$ if $C[a] \leq_1 C[b]$ for all contexts $C[\cdot]$ such that $C[a]$ and $C[b]$ are closed. Define a to be *observationally congruent* to b, or $a \sim_C b$, if $a \leq_C b$ and $b \leq_C a$.

Observational congruence need not be the same as $\sim$ in an extensional lcs. First, the lcs might not have "enough destructors". For example, in Λ without ap but with an additional 0-ary canonical operator θ, the terms $\lambda x.\, \theta$ and $\lambda x.\, \lambda y.\, \theta$ are $\sim_C$ but not $\sim$. Secondly, $\sim$ and $\sim_C$ will usually differ in the presence of nondeterminism. For example, consider an lcs that contains Λ, the integers and an operation amb such that $amb(a, b) \rightarrow c$ if and only if $a \rightarrow c$ or $b \rightarrow c$. Then $amb(\lambda x.\, 0, \lambda x.\, 1)$ and $\lambda x.\, amb(0, 1)$ are $\sim_C$ but not $\sim$. An argument can be made, however, that the bisimulation equivalence $\sim$ is more appropriate for nondeterministic programs than is the trace equivalence $\sim_C$. There does not yet seem to be a consensus on this issue. (See, for example, [11] and [7].)

It is not very surprising that if we rule out nondeterminism and ensure that there are enough destructors, then $\sim$ is the same as $\sim_C$. We make this precise as follows.

Let θ be a canonical operator, and let $\langle k_1, \ldots, k_n \rangle$ be its arity. $\mathcal{S}$ is *computationally adequate* for θ if for each i, $1 \leq i \leq n$, and for all sequences of closed terms $a_1, \ldots, a_{k_i}$ there is a context $C[\cdot]$ such that for all closed $\theta(\overline{t})$, if the i^{th} component of $\overline{t}$ is $x_1, \ldots, x_{k_i}.\, t$, then $C[\theta(\overline{t})]$ is closed and

$$C[\theta(\overline{t})] \sim t[a_1, \ldots, a_{k_i}/x_1, \ldots, x_{k_i}].$$

$\mathcal{S}$ is *computationally adequate* if it is computationally adequate for each of its canonical operators. Finally, $\mathcal{S}$ is *deterministic* if its evaluation relation is deterministic.

Theorem 3 *If $\mathcal{S}$ is a deterministic and computationally adequate extensional lcs, and if $a \leq_C b$ implies that $\sigma(a) \leq_C \sigma(b)$ for any substitution σ such that $\sigma(a)$ and $\sigma(b)$ are closed, then for all $a, b \in T(L)$, $a \leq_C b \Leftrightarrow a \leq b$.*

The proof is straightforward. The condition that $\leq_C$ be preserved under closing substitutions is satisfied if $\mathcal{S}$ contains Λ. It is easy to show that Λ satisfies the hypotheses of Theorem 3.

4 Application To Type Theory

We briefly discuss the application of the preceding results to Martin-Löf's type theory [10]. We will henceforth call this type theory ITT (for "intuitionistic type theory"). The application to Nuprl's type theory is similar. In what follows we assume some familiarity with ITT.

At the base of ITT is an untyped computation system. A deterministic lcs is obtained directly from this computation system by taking $\xrightarrow{k}$ to be the restriction of evaluation to k reduction steps. The verifications that the operators of this lcs are extensional are simple, being very similar to the argument given for application in Λ. This lcs is not computationally adequate since there are no forms for analyzing types. A computationally adequate lcs is obtained by either deleting the types or by adding a universe elimination form (which is in fact semantically sensible if type equality is taken to be intensional).

The most important fact about $\sim$ in ITT is that it is respected by the type system: if T is a type and $T \leq T'$ then T' is a type and $T = T'$, and if $t \in T$ and $t \leq t'$ then $t = t' \in T$. This fact can be proven by induction on the construction of the ITT semantics given in [2]. We will not give the argument here; instead, we will just sketch the portion of it that deals with function types. Suppose then that T is a type such that $T \rightarrow \Pi\, x : A\,.\, B$, and that $T \leq T'$. Since $\leq \subset [\leq]$, there are A' and B' such that $T' \rightarrow \Pi\, x : A'\,.\, B'$, $A \leq A'$ and $B \leq B'$. By induction, A' is a type and $A = A'$. Consider $a = a' \in A'$. $B[a]$ and $B[a']$ are (equal) types, $B[a] \leq B'[a]$ and $B[a'] \leq B'[a']$, so by induction $B'[a]$ and $B'[a']$ are types with $B[a] = B'[a]$ and $B[a'] = B'[a']$. Thus $B'[a] = B'[a']$, and so T' is a type which is equal to T.

Suppose now that $t \in \Pi\, x : A \,.\, B$ and $t \leq t'$. For some b, $t \longrightarrow \lambda x.\, b$, so there is a b' such that $t' \longrightarrow \lambda x.\, b'$ and $b \leq b'$. For each $a \in A$ we have $b[a] \leq b'[a]$ so by induction $b[a] = b'[a] \in B[a]$. It follows that $\lambda x.\, b = \lambda x.\, b' \in \Pi\, x : A \,.\, B$.

Evaluation in ITT is based on reduction; it proceeds by successively replacing redices by their contracta. Since the redex-contractum relation is contained in $\sim$, it follows that in reasoning about types and their members we can freely replace redices by contracta (and *vice versa*). This was previously unknown, although a complicated approximation to it was developed for Nuprl [8]. Experience with Nuprl (see [9] for example) indicates that this extension will be valuable in practice.

Since $\sim$ seems to be an intuitively appealing equality, at least for a deterministic extensional lcs, it is reasonable to add a type to ITT to represent it. This would open the way to inclusion in implementations of ITT such classical theorem-proving procedures as congruence closure [12].

Acknowledgements

We owe special thanks to Stuart Allen for his substantial suggestions regarding the presentation of these results.

References

[1] S. Abramsky. The lazy lambda calculus. Proceedings of the Institute of Declarative Programming, August 1987.

[2] S. F. Allen. A non-type theoretic definition of Martin-Löf's types. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 215–221. IEEE, 1987.

[3] S. F. Allen. A non-type-theoretic semantics for type-theoretic language. Technical Report 87-866, Department of Computer Science, Cornell University, September 1987. Ph.D. Thesis.

[4] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[5] Berry. Some syntactic and categorical constructions of lambda-calculus models. Technical Report 80, I.N.R.I.A., 1981.

[6] B. Bloom. Can LCF be topped? flat lattice models of typed lambda calculus. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 282–295. IEEE, 1988.

[7] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced: Preliminary report. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 229–239. IEEE, 1988.

[8] R. L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[9] D. J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* PhD thesis, Cornell University, 1988.

[10] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[11] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1980.

[12] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, 1980.