Simply-typed Underdeterminism

Ewen Denney*

LFCS, Department of Computer Science, University of Edinburgh
Kings Buildings, Edinburgh EH9 3JZ, Scotland
Email : ewd@dcs.ed.ac.uk

February 26, 1998

Abstract

We study an extension of the simply-typed lambda calculus with constructs for expressing a notion we call *underdeterminism*. This allows us to interpret notions of stub and skeleton used in top-down program development. We axiomatise a simple notion of program refinement, and give a semantics, for which the calculus is proved sound and complete.

1 Introduction

When programmers write programs they often use a mixture of top-down and bottom-up development. Central to the method of top-down development is the idea of a *stub*. Suppose we are writing some program, the main body of which needs to use a sorting function on lists. We prefer to get the structure of the main body correct first before writing the sorting function, but in order to get the program to compile, or even to type-check, we must at least declare the sorting function, and give it a dummy body — a stub.

$$\texttt{fun Sort l} = \texttt{nil};$$

Any use of the function elsewhere is now well-defined

$$\texttt{l}' = \texttt{Sort some\_list};$$

Conceptually, however, it is inelegant to use dummy code. Practically, although the program will compile and will run, the programmer must always bear in mind that this is not 'true' code. When testing the behaviour of the partially written program this must be taken into account. One way of avoiding this would be to use some kind of error, or exception mechanism as a stub. This is not entirely satisfactory however, since the programmer might forget that various stubs are left throughout the program. It would be better to have a language construct that allows the programmer to write what we might call *true stubs*. The sorting function would be written as

$$\texttt{fun Sort l} = ?_{\texttt{int list}};$$

The idea is that $?_{\texttt{int list}}$ stands for some unwritten code of type `int list`. This can then be type-checked and the system would give a warning message to the effect that certain lines of code still remain to be implemented. We would like to be able to compile and run such partially written programs. In general, this is not possible since our program might just be one big stub. However, we would like an understanding of programs with stubs which lets us do this whenever possible.

*Research supported by a grant from the Engineering and Physical Sciences Research Council

A similar concept used in program development is that of program *skeletons*. These are templates of code expressing useful algorithm schemes, such as divide-and-conquer. Using the $?_\tau$ construct, it is possible to define a simple notion of skeleton[1] within the language, rather than as some kind of meta-entity. As explained below, we cannot in general express stubs and skeletons using parameterisation, since parameters cannot range over code which can contain local variables.

In this paper, we study an extension of the simply-typed lambda calculus, $\lambda^{\times\rightarrow}$, with constructs $?_\tau$ for each type τ, giving a calculus $\lambda_?$. We give an equational theory for $\lambda_?$ and we study the natural notion of refinement induced by this.

We can compile pure functional programming languages into the simply-typed lambda calculus and, similarly, can compile a language with *true stubs* into $\lambda_?$. The theory of $\lambda_?$ could be used to prove the correctness of transformations carried out by a compiler.

It is possible to extend the calculus to a full refinement calculus, in which stubs contain logical assumptions. An equational theory is also necessary in order to give a full program logic for this extended language. This is carried out in the author's forthcoming PhD thesis. However, we believe it is worthwhile to study this simpler system, for both theoretical interest and for the practical applications described above.

2 Overview

We introduce an extension of the simply-typed lambda calculus ($\lambda^{\times\rightarrow}$) with constructs for expressing a notion we call *underdeterminism*. Consider the term $\lambda x : \sigma.\langle 2, x\rangle$ of type $\sigma \rightarrow \mathtt{nat} \times \sigma$. This term is *determined* in the sense that we have complete knowledge about it. Suppose now, we know that some term of the same type always returns pairs of which the left component is 2, but we know nothing about the right component. We might write this as $\lambda x : \sigma.\langle 2, ?_\sigma\rangle$, where $?_\sigma$ means 'some term of type σ'. We allow $?_\sigma$ to stand for any subterm which is well-formed in the local context, so these unknowns can contain the variable x. This is in contrast to the use of variables as indeterminates. If, instead of the stub $?_\sigma$, we use a free variable y of type σ as a parameter, writing the above term as $\lambda x : \sigma.\langle 2, y\rangle$, then we can substitute any term for y which is well-formed in the global context. The point is, though, that because of variable capture, we cannot substitute x to get $\lambda x : \sigma.\langle 2, x\rangle$. This, of course, is crucial to the use of stubs.

Now, a less determined term would be $\lambda x : \sigma.\langle ?_{\mathtt{nat}}, ?_\sigma\rangle$. This is a term of type $\sigma \rightarrow (\mathtt{nat} \times \sigma)$ which returns ... what? Evaluation does not make sense in general for such partially constructed terms anyway, but our intuition tells us that the term $\lambda x : \sigma.\langle ?_{\mathtt{nat}}, ?_\sigma\rangle$ carries the same information as $\lambda x : \sigma.?_{\mathtt{nat}\times\sigma}$ and $?_{\sigma\rightarrow(\mathtt{nat}\times\sigma)}$. We would like to prove equivalences such as this, and in general consider a specialisation ordering at each type, such that

$$?_{\sigma\rightarrow(\mathtt{nat}\times\sigma)} \;\sqsubseteq\; \lambda x : \sigma.\langle 2, ?_\sigma\rangle \;\sqsubseteq\; \lambda x : \sigma.\langle 2, x\rangle$$

Moreover, we would like to study how underdeterminism interacts with the usual equational rules of $\lambda^{\times\rightarrow}$. Our interest in such a calculus of refinement comes from our belief that this is a fundamental aspect of program development. The refinement methodology of program development consists of writing a term meaning 'a program which satisfies specification ϕ', and transforming it step by step into an actual program satisfying the specification ϕ. We view types as rudimentary specifications, and defer study of the logic to a separate report [2]. We believe it is worthwhile to study underdeterminism in isolation from logic, as much of the difficulty in reasoning in refinement calculi is in understanding how underdeterminism interacts with the programming features.

In Section 3 we describe the language and its refinement rules. The equational theory is then studied as part of a simple logic. We give a simple denotational semantics in Section 4 as a class of set-theoretic models, and show that our calculus is complete for proving the refinements valid in this class of models.

[1] A full account of skeletons would make reference to software architecture and is outside the scope of this paper. The simple characterisation given here seems a reasonable starting point though.

In Section 5 we describe a first-order logic of refinement and give its semantics. The logic is also proven sound and complete. Finally, we draw some brief conclusions in Section 6.

Related Work

The study of indefinite descriptions — 'some x such that P' — goes right back to the earliest work on modern logic (*e.g.* [9]) but the idea there is that a description ranges over semantic values. Similarly, Hermida and Jacobs' study of indeterminates in the lambda calculus [4] does not account for substitution allowing variable capture.

Although much work has been published on refinement calculi, there seem to be no fully axiomatised systems. [8] describes the classical refinement calculus of Back, Morgan and Morris. This is an imperative language extended with specification constructs. Their system however, uses *nondeterminism* to express specification constructs. We believe this to be a mistake, as nondeterminism is a computational phenomenon distinct from our view of underdeterminism as a specificational phenomenon, at a level above the programming language. Moreover, we might want to consider a combination of nondeterminism and underdeterminism, for example when developing a logic program.

Proof development systems, such as Lego [5], allow users to interactively construct a proof by refinement. Intermediate states in a proof development may be modelled as underdetermined terms. The notion of metavariable or logical or existential variable is similar to our ? terms.

The concept of underdeterminism also arises in linguistics (with the name 'underspecification'), where semantically ambiguous statements such as "every student is $?_{\texttt{noun\_phrase}}$" are studied. [1] for example, considers a language with metavariables for representing such statements.

3 The Calculus

We give the syntax of the language and the classes of judgements. Then we give some syntactic results and a short example of refinement.

3.1 Syntax

We extend the simply-typed lambda calculus with an underdeterminism construct, $?_\tau$, meaning 'some term with type τ'. We view the type τ as a rudimentary specification and refer to terms with such 'holes' for programs to be supplied later, as *refinement terms*. Here "simply-typed lambda calculus" means with finite products and an applied theory of ground types γ and constants k. The types, preterms and environments are

$$\tau ::= \mathbf{1} \mid \gamma \mid \tau \times \tau \mid \tau \to \tau$$

$$r ::= x \mid k(r_1, \ldots, r_n) \mid * \mid \langle r, r' \rangle \mid \lambda x : \tau . r \mid ?_\tau \mid \pi_1(r) \mid \pi_2(r) \mid rr' \mid \texttt{let}\ x : \tau\ \texttt{be}\ r\ \texttt{in}\ r'$$

$$\Gamma ::= \langle \rangle \mid \Gamma, x : \tau$$

We say that a term is *determined* if it contains no stubs, that is, subterms of the form $?_\tau$. Otherwise, we call a term *underdetermined*. We use the metavariable t to range over determined terms, and r over arbitrary underdetermined terms. Note that primitive constants are determined. In fact, each term of the simply-typed lambda calculus is determined. The converse does not hold, however, since we allow $\texttt{let}\ x : \sigma\ \texttt{be}\ t\ \texttt{in}\ t'$ to be determined. We will see, though, that by eliminating $\texttt{let}$'s, every determined term is provably equal to an ordinary term of the simply-typed lambda calculus. This is since $\texttt{let}\ x : \sigma\ \texttt{be}\ t\ \texttt{in}\ t'$ equals $t'[t/x]$.

We use the notation $r[t/x]$ to indicate the (capture avoiding) substitution of t for each free occurrence of variable x in r. We sometimes write $r[x]$ to distinguish all the free occurrences of variable x in term r. This does not mean that x is the only free variable, nor that it actually appears free in r. When writing $r[x]$, we can use $r[t]$ to indicate the substitution $r[t/x]$.

We assume throughout some fixed signature of types and constants.

Definition 3.1 *A* $\lambda_?$-signature *consists of a collection of ground types* γ *and constants* k*, each of which is assigned some sort* $\tau_1, \ldots, \tau_n \to \tau$.

We write $k : \tau_1, \ldots, \tau_n \to \tau$ to indicate that constant k has sort $\tau_1, \ldots, \tau_n \to \tau$. This does not mean that k itself is a well-typed term, but that with n arguments, $r_1 : \tau_1$, ..., $r_n : \tau_n$, the term $k(r_1, \ldots, r_n)$ has type τ.

Intuitively, we can think of a term r as being a kind of description or specification of determined terms, so for example, $\langle 2, ?_{\mathtt{nat}}\rangle$ is a term which specifies pairs of naturals $\langle 2, t\rangle$, where t is any natural. Conversely, we say that $\langle 2, t\rangle$ satisfies $\langle 2, ?_{\mathtt{nat}}\rangle$.

The abstraction $\lambda x : \tau.r$ should be thought of as describing the collection of determined functions $\lambda x : \tau.t$, such that for all determined arguments t' of type τ, $t[t'/x]$ satisfies $r[t'/x]$, and not as a nondeterministic function which takes an argument t' and returns a term $r[t'/x]$.

We want to regard variables as ranging over determined terms, in order to retain the familiar rules of $\lambda^{\times\to}$ for determined terms. Since only determined terms can be substituted for variables, we use an axiomatisation of `let` expressions, $\mathtt{let}\ x : \tau\ \mathtt{be}\ r\ \mathtt{in}\ r'$, as in the computational lambda calculus [7], as a way of discharging an arbitrary underdetermined term r, without substituting directly for a variable x. The idea is that $\mathtt{let}\ x : \tau\ \mathtt{be}\ r\ \mathtt{in}\ r'$ delays the substitution of r for x until r can be refined into some determined t, but still lets us reason about the substitution. The expression $\mathtt{let}\ x : \tau\ \mathtt{be}\ r\ \mathtt{in}\ r'$ then, can be satisfied by t', when t satisfies $r'[t/x]$ for some t which satisfies r.

Although we do not assume that all types are inhabited, for various statements below we will make this restriction. This is a natural assumption when types are used in programming and avoids certain semantic complications (see [6]). The reader may assume though that this assumption means that our calculus is of no interest in studying program specification, where in traditional type-theoretic approaches, specifications are viewed as possibly empty types. However, the idea in [2] is not to use types themselves as specifications, but that a specification (which may be unsatisfiable) is something 'over' an ordinary program type. *e.g.* The specification $(n : \mathtt{nat})\mathtt{even}(n)$ (in the notation of [2]) is over `nat`.

3.2 Judgements

We axiomatise an equational theory with judgements

Typing	$\Gamma \vdash r : \tau$
Refinement	$\Gamma \vdash r \sqsubseteq_\tau r'$

where Γ is a context of variable assumptions. As is usual with lambda calculi, (in)equations are at a type, which we sometimes drop when not significant. We write the refinement of r to r' (at τ) as $r \sqsubseteq_\tau r'$, to indicate that r' is more determined than r. Note that some authors use $\sqsupseteq$ for refinement.

We take equality to be a derived notion, defined as mutual refinement. This is reasonable because, as we show by a semantic argument below, the calculus is a conservative extension of the simply-typed lambda calculus, *i.e.* determined terms are mutually refinable if and only if they are provably equal in $\lambda^{\times\to}$.

Contexts are well-formed

$$\vdash \Gamma\ \mathtt{wf}$$

when they contain distinct variables. We adopt the convention that in writing a judgement we assume its context to be well-formed.

We assume throughout some fixed applied theory given (with respect to a signature) as a set of well-typed equations in context between determined terms.

Definition 3.2 *A* $\lambda_?$-theory *consists of a* $\lambda_?$*-signature, and a collection of equations in context,* $\Gamma \vdash t =_\tau t'$*, well-typed with respect to the signature.*

We write $Ax \triangleright \Gamma \vdash t =_\tau t'$ to indicate that an equation is an axiom in the applied theory.

Example 3.1 We give an axiomatisation of booleans, as an example of an applied theory. There is one ground type

$$\gamma ::= \texttt{bool}$$

and three constant symbols

$$\texttt{true} : \texttt{bool}$$

$$\texttt{false} : \texttt{bool}$$

$$\texttt{if \_ then \_ else \_} : \texttt{bool}, \tau, \tau \to \tau$$

Strictly speaking, there are separate conditionals for each type τ but we ignore this.

The axioms are given in a schematic form for arbitrary contexts :

$$\Gamma \vdash \texttt{if true then } x \texttt{ else } y =_\tau x \qquad (x : \tau, y : \tau \in \Gamma)$$

$$\Gamma \vdash \texttt{if false then } x \texttt{ else } y =_\tau y \qquad (x : \tau, y : \tau \in \Gamma)$$

$$\Gamma \vdash f(b) =_\tau \texttt{if } b \texttt{ then } f(\texttt{true}) \texttt{ else } f(\texttt{false}) \qquad (f : \texttt{bool} \to \tau, b : \texttt{bool} \in \Gamma)$$

The equations are between determined terms. Refinements follow from the rules of the calculus.

3.3 Rules of the equational calculus

The typing judgement $\Gamma \vdash r : \tau$ is axiomatised in Figure 1. This just extends the rules of the simply-typed lambda calculus with typing rules for the ? and `let` constructs.

The equality rules in Figure 2 are just those of $\lambda^{\times\to}$, and should be read as mutual refinements. The rules are given for determined terms. Although we show below various generalisations of these to arbitrary underdetermined terms, we prefer to give the axioms of the calculus in this minimal form as it more clearly shows that refinement is an axiomatisation on top of the underlying equational theory.

Figure 3 axiomatises how underdeterminism combines with program constructs via the `let` expressions. Most of these rules are borrowed from the computational lambda calculus. The exception is (LET-ABS-EQ), which makes explicit the 'hidden dependency' of specifications on variables in the context. In view of (LET-ASSOC-EQ), we use $\texttt{let } x_1 : \sigma_1, x_2 : \sigma_2 \texttt{ be } r_1, r_2 \texttt{ in } r$ as an abbreviation of the nested `let` expression $\texttt{let } x_1 : \sigma_1 \texttt{ be } r_1 \texttt{ in } (\texttt{let } x_2 : \sigma_2 \texttt{ be } r_2 \texttt{ in } r)$. Note that because of the assumption of well-formedness of contexts, we can omit side-conditions on the occurrence of variables. For example, in (LET-ASSOC-EQ), since $\Gamma, y : \tau$ is well-formed, y is not in Γ and so y is not in r''.

Figures 4 and 5 axiomatise the refinement relation. The intuition behind the refinement relation is that it should correspond to an increase in information, and a decrease in the possible programs which could satisfy a refinement term. There are top-down rules for decomposing a specification by refinement into a combination of simpler ones. We also have a weakening rule (LET-WEAKEN-REF) which may be thought of as claiming an auxiliary lemma, and a congruence rule (LET-ξ-REF) for `let` expressions, which lets us derive the corresponding rules for pairs, applications and projections. Sequent style bottom-up rules for making use of the context are derived below.

The following lemma provides some insight into underdeterminism. It says that in any term, all the underdeterminism can be moved to the outside, so the term can be viewed as a simple combination of program and specification. For example, $\lambda x : \sigma.\langle 2, ?_{\texttt{nat}}\rangle$ is equal to $\texttt{let } f : \sigma \to \texttt{nat be } ?_{\sigma\to\texttt{nat}} \texttt{ in } \lambda x : \sigma.\langle 2, fx\rangle$.

Lemma 3.1 *For all terms in context $\Gamma \vdash r : \tau$, there exists a context $x_1 : \sigma_1, \ldots, x_n : \sigma_n$ and a determined term $\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash t : \tau$ such that each x_i appears exactly once in t, and $\Gamma \vdash \texttt{let } x_1 : \sigma_1, \ldots, x_n : \sigma_n \texttt{ be } ?_{\sigma_1}, \ldots, ?_{\sigma_n} \texttt{ in } t =_\tau r$.*

Proof: Use `let` rules to move the underdeterminism outwards, (LET-ABS-EQ) handling the abstractions. ∎

$$\Gamma, x:\sigma, \Gamma' \vdash x:\sigma \qquad \text{(VAR-TYPE)}$$

$$\frac{\Gamma \vdash r_i : \tau_i \quad (i = 1, \ldots, n)}{\Gamma \vdash k(r_1, \ldots, r_n) : \tau} \quad (k : \tau_1, \ldots, \tau_n \to \tau) \qquad \text{(CONST-TYPE)}$$

$$\Gamma \vdash * : \mathbf{1} \qquad \text{(UNIT-TYPE)}$$

$$\Gamma \vdash ?_\sigma : \sigma \qquad \text{(SPEC-TYPE)}$$

$$\frac{\Gamma \vdash r : \sigma \quad \Gamma \vdash r' : \tau}{\Gamma \vdash \langle r, r' \rangle : \sigma \times \tau} \qquad \text{(PAIR-TYPE)}$$

$$\frac{\Gamma, x : \sigma \vdash r : \tau}{\Gamma \vdash \lambda x : \sigma . r : \sigma \to \tau} \qquad \text{(ABS-TYPE)}$$

$$\frac{\Gamma \vdash r : \sigma \times \tau}{\Gamma \vdash \pi_1(r) : \sigma} \qquad \text{(PROJ1-TYPE)}$$

$$\frac{\Gamma \vdash r : \sigma \times \tau}{\Gamma \vdash \pi_2(r) : \tau} \qquad \text{(PROJ2-TYPE)}$$

$$\frac{\Gamma \vdash r : \sigma \to \tau \quad \Gamma \vdash r' : \sigma}{\Gamma \vdash r r' : \tau} \qquad \text{(APP-TYPE)}$$

$$\frac{\Gamma \vdash r : \tau \quad \Gamma, x : \tau \vdash r' : \tau'}{\Gamma \vdash \texttt{let}\ x : \tau\ \texttt{be}\ r\ \texttt{in}\ r' : \tau'} \qquad \text{(LET-TYPE)}$$

Figure 1: Typing Rules

Note the linearity – each stub counts exactly once, so $\langle ?_\sigma, ?_\sigma \rangle$ becomes $\texttt{let}\ x : \sigma, y : \sigma\ \texttt{be}\ ?_\sigma, ?_\sigma\ \texttt{in}\ \langle x, y \rangle$ and not $\texttt{let}\ z : \sigma\ \texttt{be}\ ?_\sigma\ \texttt{in}\ \langle z, z \rangle$ (to which it refines though). We will not, however, use the linearity in appeals to the lemma below. Note also that such canonical forms need not be unique. We sometimes use the useful abbreviation $\texttt{let}\ x_1 : \sigma_1, \ldots, x_n : \sigma_n\ \texttt{in}\ t$ for this canonical form. So, for example, $\lambda x : \sigma . \langle 2, ?_{\texttt{nat}} \rangle$ is equal to $\texttt{let}\ f : \sigma \to \texttt{nat}\ \texttt{in}\ \lambda x : \sigma . \langle 2, fx \rangle$. In fact, by repeated pairing, we can always express terms in the simpler form $\texttt{let}\ x : \sigma\ \texttt{in}\ t$.

It is often convenient to write $\texttt{let}\ x_1 : \sigma_1, \ldots, x_n : \sigma_n\ \texttt{in}\ t$ as $\texttt{let}\ \Gamma\ \texttt{in}\ t$, where Γ stands for the local context $x_1 : \sigma_1, \ldots, x_n : \sigma_n$. We also write $t[\Gamma]$ as informal notation for $t[x_1 : \sigma_1, \ldots, x_n : \sigma_n]$.

By expressing terms in canonical form, we can prove a few results about `let` expressions. None of these results hold in the computational lambda calculus.

Lemma 3.2 *The following rules are derived :*

1. *(*`let`*-commutativity)*

$$\frac{\Gamma \vdash r : \tau \qquad \Gamma \vdash r' : \tau' \qquad \Gamma, x : \tau, y : \tau' \vdash r'' : \tau''}{\Gamma \vdash \texttt{let}\ x : \tau, y : \tau'\ \texttt{be}\ r, r'\ \texttt{in}\ r'' =_{\tau''} \texttt{let}\ y : \tau', x : \tau\ \texttt{be}\ r', r\ \texttt{in}\ r''}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau \quad \Gamma \vdash t' : \sigma}{\Gamma \vdash (\lambda x : \sigma.t)t' =_\tau t[t'/x]} \qquad \text{(ABS-}\beta\text{-EQ)}$$

$$\frac{\Gamma \vdash t : \sigma \to \tau}{\Gamma \vdash \lambda x : \sigma.(tx) =_{\sigma\to\tau} t} \qquad \text{(ABS-}\eta\text{-EQ)}$$

$$\frac{\Gamma \vdash t_1 : \sigma_1 \quad \Gamma \vdash t_2 : \sigma_2}{\Gamma \vdash \pi_i\langle t_1, t_2\rangle =_{\sigma_i} t_i} \ (i = 1, 2) \qquad \text{(PAIR-}\beta\text{-EQ)}$$

$$\frac{\Gamma \vdash t : \sigma \times \tau}{\Gamma \vdash \langle \pi_1(t), \pi_2(t)\rangle =_{\sigma\times\tau} t} \qquad \text{(PAIR-}\eta\text{-EQ)}$$

$$\frac{\Gamma \vdash t : \mathbf{1}}{\Gamma \vdash t =_{\mathbf{1}} *} \qquad \text{(UNIT-}\eta\text{-EQ)}$$

$$\Gamma \vdash t =_\tau t' \qquad (Ax \rhd \Gamma \vdash t =_\tau t') \quad \text{(AXIOM-EQ)}$$

Figure 2: Equality rules for determined terms

$$\frac{\Gamma \vdash t : \tau \quad \Gamma, x : \tau \vdash r : \tau'}{\Gamma \vdash \texttt{let } x : \tau \texttt{ be } t \texttt{ in } r[x] =_{\tau'} r[t]} \qquad \text{(LET-}\beta\text{-EQ)}$$

$$\frac{\Gamma \vdash r : \tau}{\Gamma \vdash \texttt{let } x : \tau \texttt{ be } r \texttt{ in } x =_\tau r} \qquad \text{(LET-}\eta\text{-EQ)}$$

$$\frac{\Gamma \vdash r : \tau \quad \Gamma, y : \tau \vdash r' : \tau' \quad \Gamma, x : \tau' \vdash r'' : \tau''}{\begin{array}{l}\Gamma \vdash \texttt{let } x : \tau' \texttt{ be } (\texttt{let } y : \tau \texttt{ be } r \texttt{ in } r'[y]) \texttt{ in } r''[x] \\ \quad =_{\tau''} \texttt{let } y : \tau \texttt{ be } r \texttt{ in } (\texttt{let } x : \tau' \texttt{ be } r'[y] \texttt{ in } r''[x])\end{array}} \qquad \text{(LET-ASSOC-EQ)}$$

$$\frac{\Gamma \vdash r_i : \tau_i}{\begin{array}{l}\Gamma \vdash \texttt{let } x_1 : \tau_1, \ldots, x_n : \tau_n \texttt{ be } r_1, \ldots, r_n \\ \quad \texttt{in } k(x_1, \ldots, x_n) =_\tau k(r_1, \ldots, r_n)\end{array}} \ (k : \tau_1, \ldots, \tau_n \to \tau) \qquad \text{(LET-CONST-EQ)}$$

$$\frac{\Gamma \vdash r : \sigma \to \tau \quad \Gamma \vdash r' : \sigma}{\Gamma \vdash \texttt{let } x : \sigma \to \tau, x' : \sigma \texttt{ be } r, r' \texttt{ in } xx' =_\tau rr'} \qquad \text{(LET-APP-EQ)}$$

$$\frac{\Gamma \vdash r : \tau \quad \Gamma \vdash r' : \tau'}{\Gamma \vdash \texttt{let } x : \tau, x' : \tau' \texttt{ be } r, r' \texttt{ in } \langle x, x'\rangle =_{\tau\times\tau'} \langle r, r'\rangle} \qquad \text{(LET-PAIR-EQ)}$$

$$\frac{\Gamma \vdash r : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{let } x : \tau_1 \times \tau_2 \texttt{ be } r \texttt{ in } \pi_i(x) =_{\tau_i} \pi_i(r)} \qquad \text{(LET-PROJi-EQ)}$$

$$\frac{\Gamma, x : \sigma, y : \tau \vdash r[x, y] : \tau'}{\begin{array}{l}\Gamma \vdash \texttt{let } z : \sigma \to \tau \texttt{ be } ?_{\sigma\to\tau} \texttt{ in } \lambda x : \sigma.r[x, zx] \\ \quad =_{\sigma\to\tau'} \lambda x : \sigma.(\texttt{let } y : \tau \texttt{ be } ?_\tau \texttt{ in } r[x, y])\end{array}} \qquad \text{(LET-ABS-EQ)}$$

Figure 3: `let` equalities

$$\Gamma, x : \sigma, \Gamma' \vdash ?_\sigma \sqsubseteq_\sigma x \qquad \text{(VAR-REF)}$$

$$\Gamma \vdash ?_\tau \sqsubseteq_\tau k(?_{\tau_1}, \ldots, ?_{\tau_n}) \quad (k : \tau_1, \ldots, \tau_n \to \tau) \qquad \text{(CONST-REF)}$$

$$\Gamma \vdash ?_{\mathbf{1}} \sqsubseteq_{\mathbf{1}} * \qquad \text{(UNIT-REF)}$$

$$\Gamma \vdash ?_{\sigma\times\tau} \sqsubseteq_{\sigma\times\tau} \langle ?_\sigma, ?_\tau \rangle \qquad \text{(PAIR-REF)}$$

$$\Gamma \vdash ?_{\sigma\to\tau} \sqsubseteq_{\sigma\to\tau} \lambda x : \sigma.?_\tau \qquad \text{(ABS-REF)}$$

Figure 4: Top-down refinement rules

$$\frac{\Gamma \vdash r_1 \sqsubseteq_{\tau_1} r_1' \quad \Gamma, x : \tau_1 \vdash r_2 \sqsubseteq_{\tau_2} r_2'}{\Gamma \vdash \mathtt{let}\ x : \tau_1\ \mathtt{be}\ r_1\ \mathtt{in}\ r_2 \sqsubseteq_{\tau_2} \mathtt{let}\ x : \tau_1\ \mathtt{be}\ r_1'\ \mathtt{in}\ r_2'} \qquad \text{(LET-}\xi\text{-REF)}$$

$$\frac{\Gamma, x : \sigma \vdash r \sqsubseteq_\tau r'}{\Gamma \vdash \lambda x : \sigma.r \sqsubseteq_{\sigma\to\tau} \lambda x : \sigma.r'} \qquad \text{(ABS-}\xi\text{-REF)}$$

$$\frac{\Gamma \vdash r : \tau}{\Gamma \vdash r \sqsubseteq_\tau r} \qquad \text{(REFL-REF)}$$

$$\frac{\Gamma \vdash r \sqsubseteq_\tau r' \quad \Gamma \vdash r' \sqsubseteq_\tau r''}{\Gamma \vdash r \sqsubseteq_\tau r''} \qquad \text{(TRANS-REF)}$$

$$\frac{\Gamma \vdash r' : \tau \quad \Gamma \vdash r : \sigma}{\Gamma \vdash r' \sqsubseteq_\tau \mathtt{let}\ x : \sigma\ \mathtt{be}\ r\ \mathtt{in}\ r'}\ x \notin r' \qquad \text{(LET-WEAKEN-REF)}$$

Figure 5: Other refinement rules

2. (let-*contraction)*

$$\frac{\Gamma \vdash r : \sigma \qquad \Gamma, x : \sigma, y : \sigma \vdash r' : \tau}{\Gamma \vdash \mathtt{let}\ x : \sigma, y : \sigma\ \mathtt{be}\ r, r\ \mathtt{in}\ r' \sqsubseteq_\tau \mathtt{let}\ z : \sigma\ \mathtt{be}\ r\ \mathtt{in}\ r'[z/x, z/y]}$$

3. *(Strengthen local context)*
   For $x \notin r'$,

$$\frac{\Gamma \vdash r' : \tau' \qquad \Gamma, x : \tau, y : \tau' \vdash r : \tau''}{\Gamma \vdash \lambda x : \tau.(\mathtt{let}\ y : \tau'\ \mathtt{be}\ r'\ \mathtt{in}\ r) \sqsubseteq_{\tau \to \tau''} \mathtt{let}\ y : \tau'\ \mathtt{be}\ r'\ \mathtt{in}\ \lambda x : \tau.r}$$

Proof: We prove (3).
First we show that

$$\lambda x : \sigma.\mathtt{let}\ y : \tau\ \mathtt{in}\ r[y] \sqsubseteq \mathtt{let}\ y : \tau\ \mathtt{in}\ \lambda x : \sigma.r[y] \tag{4}$$

This is :

$$\begin{array}{lll}
\multicolumn{2}{l}{\lambda x : \tau.\mathtt{let}\ y : \tau'\ \mathtt{in}\ r[y]} & \\
= & \mathtt{let}\ z : \tau \to \tau'\ \mathtt{in}\ \lambda x : \tau.r[zx] & \text{(LET-ABS-EQ)} \\
\sqsubseteq & \mathtt{let}\ y : \tau'\ \mathtt{in}\ \mathtt{let}\ z : \tau \to \tau'\ \mathtt{in}\ \lambda x : \tau.r[zx] & \text{(LET-WEAKEN-REF)} \\
\sqsubseteq & \mathtt{let}\ y : \tau'\ \mathtt{in} & (y : \tau' \vdash ?_{\tau \to \tau'} \sqsubseteq \lambda x : \tau.y, \\
 & \quad \mathtt{let}\ z : \tau \to \tau'\ \mathtt{be}\ \lambda x : \tau.y\ \mathtt{in}\ \lambda x : \tau.r[zx] & \text{and LET-}\xi\text{-REF)} \\
= & \mathtt{let}\ y : \tau'\ \mathtt{in}\ \lambda x : \tau.r[y] & \text{(LET-}\xi\text{-EQ)}
\end{array}$$

Then :

$$\begin{array}{lll}
\multicolumn{2}{l}{\lambda x : \tau.\mathtt{let}\ y : \tau'\ \mathtt{be}\ r'\ \mathtt{in}\ r[x, y]} & \\
= & \lambda x : \tau.\mathtt{let}\ y : \tau'\ \mathtt{be}\ (\mathtt{let}\ z : \sigma\ \mathtt{in}\ t)\ \mathtt{in}\ r[x, y] & \text{(Lemma 3.1; can.form)} \\
= & \lambda x : \tau.\mathtt{let}\ z : \sigma\ \mathtt{in}\ (\mathtt{let}\ y : \tau'\ \mathtt{be}\ t\ \mathtt{in}\ r[x, y]) & \text{(LET-ASSOC-EQ)} \\
\sqsubseteq & \mathtt{let}\ z : \sigma\ \mathtt{in}\ \lambda x : \tau.(\mathtt{let}\ y : \tau'\ \mathtt{be}\ t\ \mathtt{in}\ r[x, y]) & (4) \\
= & \mathtt{let}\ z : \sigma\ \mathtt{in}\ \mathtt{let}\ y : \tau'\ \mathtt{be}\ t\ \mathtt{in}\ \lambda x : \tau.r[x, y] & \text{(LET-}\beta\text{-EQ)} \\
= & \mathtt{let}\ y : \tau'\ \mathtt{be}\ (\mathtt{let}\ z : \sigma\ \mathtt{in}\ t)\ \mathtt{in}\ \lambda x : \tau.r[x, y] & \text{(LET-ASSOC-EQ)} \\
= & \mathtt{let}\ y : \tau'\ \mathtt{be}\ r'\ \mathtt{in}\ \lambda x : \tau.r[x, y] & \text{(can.form)}
\end{array}$$

The proofs of (1) and (2) are carried out similarly, by expressing terms in canonical form. ■

Commutativity of `let`'s corresponds to the idea that it does not matter what order we solve subproblems in (so long as they do not depend on each other), and contraction of `let`'s says that we can solve two identical problems by just solving the problem once and using the solution twice. The third rule illustrates the dependence of underdeterminism on the context. There are more determined terms which satisfy the term on the left, since x can be used in refining r', but this is not possible when r' is outside the bound variable on the right.

If all types are inhabited then, in fact, all terms are satisfiable. That is, every r refines to some determined t. Because of this, the rule (LET-WEAKEN-REF) can be strengthened to an equality, *i.e.* for $x \notin r'$, we have $r' = \mathtt{let}\ x : \sigma\ \mathtt{be}\ r\ \mathtt{in}\ r'$. We use this fact to derive strengthened forms of (PAIR-β-EQ) and (UNIT-η-EQ) for arbitrary underdetermined terms (of appropriate type).

Proposition 3.1 *If all types are inhabited :*

1. $\dfrac{\Gamma \vdash r_1 : \tau_1 \quad \Gamma \vdash r_2 : \tau_2}{\Gamma \vdash \pi_i\langle r_1, r_2\rangle =_{\tau_i} r_i}\ (i = 1, 2)$

2. $\dfrac{\Gamma \vdash r : \mathbf{1}}{\Gamma \vdash r =_{\mathbf{1}} *}$

Proof:

1. The first is derived as :

$$\begin{array}{rcl}\pi_1\langle r_1, r_2\rangle & = & \pi_1\langle \mathtt{let}\ x:\sigma\ \mathtt{in}\ t_1, \mathtt{let}\ y:\tau\ \mathtt{in}\ t_2\rangle \\ & = & \mathtt{let}\ x:\sigma, y:\tau\ \mathtt{in}\ \pi_1\langle t_1, t_2\rangle \\ & = & \mathtt{let}\ x:\sigma, y:\tau\ \mathtt{in}\ t_1 \\ & = & \mathtt{let}\ x:\sigma\ \mathtt{in}\ t_1 \\ & = & r_1\end{array}$$

   The $\sqsubseteq$ of the second last equality is by refining with some determined term of type τ, and the $\sqsupseteq$ is an instnace of (LET-WEAKEN-REF).

2. Suppose r has type $\mathbf{1}$. Now r has canonical form $\mathtt{let}\ x:\sigma\ \mathtt{in}\ t$, say, where t has type $\mathbf{1}$. By (UNIT-η-EQ) we can prove $x:\sigma \vdash t =_{\mathbf{1}} *$ and so $\mathtt{let}\ x:\sigma\ \mathtt{in}\ t =_{\mathbf{1}} \mathtt{let}\ x:\sigma\ \mathtt{in}\ *$, which since σ is inhabited, equals $*$.

∎

We can strengthen (ABS-β-EQ), without the assumption of nonemptiness.

Proposition 3.2 *The following can be derived :*

$$\frac{\Gamma, x:\sigma \vdash r:\tau \quad \Gamma \vdash t:\sigma}{\Gamma \vdash (\lambda x:\sigma.r)t =_\tau r[t/x]}$$

Proof: We use the auxiliary result that for any t of type σ

$$\mathtt{let}\ f:\sigma\to\tau\ \mathtt{in}\ r[ft/x] = \mathtt{let}\ x:\tau\ \mathtt{in}\ r$$

Then,

$$\begin{array}{rcl}(\lambda x:\sigma.r)t & = & (\lambda x:\sigma.\mathtt{let}\ z:\tau'\ \mathtt{in}\ t')t \\ & = & \mathtt{let}\ f:\sigma\to\tau'\ \mathtt{in}\ (\lambda x:\sigma.t'[fx/z])t \\ & = & \mathtt{let}\ f:\sigma\to\tau'\ \mathtt{in}\ t'[ft/z, t/x] \\ & = & \mathtt{let}\ z:\tau'\ \mathtt{in}\ t'[t/x] \\ & = & r[t/x]\end{array}$$

∎

Because of Proposition 3.2, we can β-reduce applications with arbitrary function bodies. This is significant as it means that underdetermined terms can be executed, up to a point, as ordinary programs. In general, this is not possible as evaluation can not proceed when a stub is encountered. For example, $\pi_1(?_{\sigma\times\tau})$ cannot be reduced. This observation forms the basis for a single-step operational semantics, although this is not investigated further here.

At this point, we pause to review our motivation for studying this calculus. Program refinement is a stepwise decomposition of logical specifications and their gradual replacment with code. The calculus which we are studying here formalises refinement for a limited form of specification, with no logic, and is a fragment of a larger calculus studied in the author's thesis.

The statement corresponding to Proposition 3.2 does not hold in the full system, as the auxiliary result fails. That is, in the presence of logic, we cannot β-reduce with arbitrary function bodies. We believe that this is motivation for studying this subcalculus of a full logical refinement calculus.

Now, Propositions 3.1 and 3.2 show that we can deduce general forms of β, although the axioms for pairs and abstractions are given for determined terms. We do not, however, have η equalities for arbitrary pairs and abstractions.

To see this for abstractions, first note, that as in the computational lambda-calculus, application distributes over `let`'s :

$$\begin{array}{rcl}(\mathtt{let}\ x\ \mathtt{be}\ r\ \mathtt{in}\ r')t & = & \mathtt{let}\ x_1\ \mathtt{be}\ (\mathtt{let}\ x\ \mathtt{be}\ r\ \mathtt{in}\ r')\ \mathtt{in}\ x_1 t \\ & = & \mathtt{let}\ x\ \mathtt{be}\ r\ \mathtt{in}\ (\mathtt{let}\ x_1\ \mathtt{be}\ r'\ \mathtt{in}\ x_1 t) \\ & = & \mathtt{let}\ x\ \mathtt{be}\ r\ \mathtt{in}\ r't\end{array}$$

Now we show that we not have $\lambda x:\sigma.rx =_{\sigma\to\tau} r$ for arbitrary underdetermined r, by proving that for $r_1 \equiv \mathtt{let}\ b:\mathtt{bool}\ \mathtt{in}\ \lambda x:\sigma.b$ and $r_2 \equiv \lambda x:\sigma.?_{\mathtt{bool}}$, we have $\lambda x:\sigma.r_1 x =_{\sigma\to\tau} r_2$.

$\lambda x:\sigma.(\texttt{let}\ b:\texttt{bool}\ \texttt{be}\ ?_{\texttt{bool}}\ \texttt{in}\ \lambda x:\sigma.b)x$
$= \quad \lambda x:\sigma.\texttt{let}\ b:\texttt{bool}\ \texttt{be}\ ?_{\texttt{bool}}\ \texttt{in}\ (\lambda x:\sigma.b)x$
$= \quad \lambda x:\sigma.?_{\texttt{bool}}$
Finally, recall that the intuition is that terms describe a set of values, and that refinement is a reduction of this set. Based on this intuition, we do not want r_1 to equal r_2. A more formal justification must await the presentation of the semantics in the next section.

We can derive inequalities, though.

Proposition 3.3 *For well-typed, r, we can derive $\lambda x:\sigma.rx \sqsubseteq_{\sigma\to\tau} r$ and $\langle \pi_1 r, \pi_2 r\rangle \sqsubseteq_{\sigma\times\tau} r$.*

Proof: We sketch the proof of the first statement. By Lemma 3.1, r has canonical form $\texttt{let}\ y:\sigma'\ \texttt{in}\ t$. Then, $\lambda x:\sigma.(\texttt{let}\ y:\sigma'\ \texttt{in}\ t)x = \lambda x:\sigma.\texttt{let}\ y:\sigma'\ \texttt{in}\ tx$, by the distribution of application over let's. By Lemma 3.2(3), this refines to $\texttt{let}\ y:\sigma'\ \texttt{in}\ \lambda x:\sigma.tx$, which by (ABS-$\eta$-EQ) and (LET-$\xi$-REF), equals $\texttt{let}\ y:\sigma'\ \texttt{in}\ t$, that is, r. ∎

The next lemma says that our refinement rules are complete, in the sense that they allow us to construct by refinement any program which satisfies a specification.

Lemma 3.3 *If $\Gamma\vdash t:\sigma$ then $\Gamma\vdash ?_\sigma \sqsubseteq_\sigma t$.*

Proof: By Proposition 3.2, $?_\sigma = (\lambda x:\sigma.?_\sigma)t$. Then $(\lambda x:\sigma.?_\sigma)t \sqsubseteq (\lambda x:\sigma.x)t = t$. ∎

The fact that this can be proven trivially motivates restricted calculi better suited to proof search, where construction must be on the structure of t. This is work in progress.

Remark 3.1 Using booleans, we can define a form of binary choice on terms. For $r, r':\sigma$, define

$$r \mathbin{[]} r' \triangleq \texttt{let}\ b:\texttt{bool}\ \texttt{in}\ (\texttt{if}\ b\ \texttt{then}\ r\ \texttt{else}\ r')$$

We can prove that $[]$ is commutative, associative, and idempotent, and so is a reasonable notion of choice. This definition is useful because it helps to illustrate the differences between underdeterminism and *nondeterminism*. If the reader is unfamiliar with nondeterminism, then this remark can be safely ignored.

We compare our axiomatisation of underdeterminism with a notion of external nondeterminism, that is, where the nondeterminism arises from the environment making the choice. In particular, we compare a nondeterministic choice operator, +, with $[]$. For example, we would intuitively expect to have $\lambda x:\texttt{nat}.2 \texttt{+} 3 = \lambda x:\texttt{nat}.2 \texttt{+} \lambda x:\texttt{nat}.3$ (using some constants for naturals). This contrasts with the properties of $[]$ under a binding, since $2 \mathbin{[]} 3$ can refine to anything well-formed in the local context. For example, $2 \mathbin{[]} 3 \sqsubseteq \texttt{if}\ x>3\ \texttt{then}\ 2\ \texttt{else}\ 3$, so $\lambda x:\texttt{nat}.2 \mathbin{[]} 3 \sqsubseteq \lambda x:\texttt{nat}.\texttt{if}\ x>3\ \texttt{then}\ 2\ \texttt{else}\ 3$.

Another difference is between nondeterministic failure terms and the analagous idea for underdeterminism — unsatisfiable terms. Suppose the type τ is uninhabited. Then there are unsatisfiable terms at every type. Any term of the form $\texttt{let}\ x:\tau\ \texttt{in}\ r$, where r is any term of type σ, will be unsatisfiable. Now, for nondeterministic failure, we expect $r \texttt{+} 0 = r$. However, if the term $\texttt{empty}_\sigma$ is unsatisfiable, then $r \mathbin{[]} \texttt{empty}_\sigma$ is also unsatisfiable. Informally, '$r \mathbin{[]} \texttt{empty}_\sigma = \texttt{empty}_\sigma$'.

We can prove the equivalences mentioned in the introduction and say that 'underdeterminism commutes with determinism', in the sense that underdeterminism at a particular type can be expressed at a lower type using the relevant term constructor. This offers some conceptual justification for regarding underdeterminism as being a feature at a level above a programming language. There is no substantial interaction with computation.

Proposition 3.4 *The following are derivable*

1. $?_{\sigma\times\tau} = \langle ?_\sigma, ?_\tau\rangle$
2. $\langle \pi_1(?_{\sigma\times\tau}), \pi_2(?_{\sigma\times\tau})\rangle = ?_{\sigma\times\tau}$

3. *If τ is inhabited, then $\pi_1(?_{\sigma\times\tau}) = ?_\sigma$, and if σ is inhabited, then $\pi_2(?_{\sigma\times\tau}) = ?_\tau$.*

4. *$?_{\sigma\to\tau} = \lambda x:\sigma.?_\tau$ and $?_{\sigma\to\tau}r =_\tau ?_\tau$ for all satisfiable $r:\sigma$.*

Proof: We prove (1), that $?_{\sigma\times\tau} =_{\sigma\times\tau} \langle ?_\sigma, ?_\tau\rangle$. Clearly, (PAIR-REF) lets us refine from left to right. We prove the other direction.

$$\begin{array}{rcl}
\langle ?_\sigma, ?_\tau\rangle & = & \texttt{let}\ x:\sigma, y:\tau\ \texttt{be}\ ?_\sigma, ?_\tau\ \texttt{in}\ \langle x,y\rangle \\
& \sqsubseteq & \texttt{let}\ z:\sigma\times\tau\ \texttt{be}\ ?_{\sigma\times\tau}\ \texttt{in} \\
& & \qquad \texttt{let}\ x:\sigma, y:\tau\ \texttt{be}\ ?_\sigma, ?_\tau\ \texttt{in}\ \langle x,y\rangle \\
& \sqsubseteq & \texttt{let}\ z:\sigma\times\tau\ \texttt{be}\ ?_{\sigma\times\tau}\ \texttt{in} \\
& & \qquad \texttt{let}\ x:\sigma, y:\tau\ \texttt{be}\ \pi_1 z, \pi_2 z\ \texttt{in}\ \langle x,y\rangle \\
& = & \texttt{let}\ z:\sigma\times\tau\ \texttt{be}\ ?_{\sigma\times\tau}\ \texttt{in}\ \langle \pi_1 z, \pi_2 z\rangle \\
& = & ?_{\sigma\times\tau}
\end{array}$$

The proofs of the other statements are carried out in a similar way, by expressing the terms in canonical form, and using the `let`-axioms and Lemma 3.2 to manipulate the terms. ∎

We can derive some useful refinement rules. There are bottom-up style refinement rules, complementing the top-down rules given as primitive, and various congruence rules.

Proposition 3.5 *The following are derivable :*

Bottom-up refinement :

$$\frac{\Gamma\vdash t:\tau \quad \Gamma, x:\tau'\vdash r\sqsubseteq_\sigma r'[x]}{\Gamma, y:\tau\to\tau'\vdash r\sqsubseteq_\sigma r'[yt]}\ x\notin r \quad \text{(APP-REF)}$$

$$\frac{\Gamma, x:\tau\vdash r\sqsubseteq_\sigma r'[x]}{\Gamma, z:\tau\times\tau'\vdash r\sqsubseteq_\sigma r'[\pi_1 z]}\ x\notin r \quad \text{(PROJ1-REF)} \qquad \frac{\Gamma, y:\tau'\vdash r\sqsubseteq_\sigma r'[y]}{\Gamma, z:\tau\times\tau'\vdash r\sqsubseteq_\sigma r'[\pi_2 z]}\ y\notin r \quad \text{(PROJ2-REF)}$$

Congruence :

$$\frac{\Gamma\vdash r_1\sqsubseteq_\sigma r_1' \quad \Gamma\vdash r_2\sqsubseteq_\tau r_2'}{\Gamma\vdash\langle r_1, r_2\rangle\sqsubseteq_{\sigma\times\tau}\langle r_1', r_2'\rangle} \quad \text{(PAIR-}\xi\text{-REF)} \qquad \frac{\Gamma\vdash r_i\sqsubseteq_{\tau_i} r_i'\ (i=1,\ldots,n)}{\Gamma\vdash k(r_1,\ldots,r_n)\sqsubseteq_\tau k(r_1',\ldots,r_n')} \quad \text{(CONST-}\xi\text{-REF)}$$

$$\frac{\Gamma\vdash r\sqsubseteq_{\tau_1\times\tau_2} r'}{\Gamma\vdash\pi_i(r)\sqsubseteq_{\tau_i}\pi_i(r')}\ (i=1,2) \quad \text{(PROJ-}\xi\text{-REF)} \qquad \frac{\Gamma\vdash r_1\sqsubseteq_{\sigma\to\tau} r_1' \quad \Gamma\vdash r_2\sqsubseteq_\sigma r_2'}{\Gamma\vdash r_1 r_2\sqsubseteq_\tau r_1' r_2'} \quad \text{(APP-}\xi\text{-REF)}$$

Proof: The bottom-up refinement rules are obtained by cutting. (PROJ1-REF), for example, is derived as :

$$\cfrac{\cfrac{\cfrac{\Gamma, x:\tau\vdash r\sqsubseteq_\sigma r'[x]}{\Gamma\vdash\lambda x:\tau.r\sqsubseteq_{\tau\to\sigma}\lambda x:\tau.r'[x]}}{\Gamma, z:\tau\times\tau'\vdash(\lambda x:\tau.r)\pi_1 z\sqsubseteq_\sigma(\lambda x:\tau.r'[x])\pi_1 z}}{\Gamma, z:\tau\times\tau'\vdash r\sqsubseteq_\sigma r'[\pi_1 z]}$$

The congruence rules all follow from (LET-ξ-REF). For example, (PAIR-ξ-REF) follows since $\langle r_1, r_2\rangle$ equals $\texttt{let}\ x_1:\tau_1, x_2:\tau_2\ \texttt{be}\ r_1, r_2\ \texttt{in}\ \langle x_1, x_2\rangle$. Then using (LET-$\xi$-REF), this refines to $\texttt{let}\ x_1:\tau_1, x_2:\tau_2\ \texttt{be}\ r_1', r_2'\ \texttt{in}\ \langle x_1, x_2\rangle$, which equals $\langle r_1', r_2'\rangle$. ∎

We make a similar point here to that made after Lemma 3.3. In a search-directed refinement calculus these rules would be primitive. Our intention here, though, was to give a system complete for proving arbitrary refinements of the form $r \sqsubseteq_\tau r'$, and not goal-directed refinements $r \sqsubseteq_\tau t$.

Example 3.2 We can use `let` weakening (or contraction) to combine two equivalent problems into one

$$\langle ?_\sigma, ?_\sigma \rangle \;\sqsubseteq\; \texttt{let}\ z : \sigma\ \texttt{be}\ ?_\sigma\ \texttt{in}\ \langle ?_\sigma, ?_\sigma \rangle \;\sqsubseteq\; \texttt{let}\ z : \sigma\ \texttt{be}\ ?_\sigma\ \texttt{in}\ \langle z, z \rangle$$

The first step uses (LET-WEAKEN-REF), and the second uses (VAR-REF), (REFL-REF), (PAIR-REF) and (LET-ξ-REF).

Example 3.3 We give a short example, using refinement to derive a swap function. Transitivity of refinement means that we can often present much of a refinement derivation as equational reasoning. Here there are two main steps.

$$?_{\sigma\times\tau\to\tau\times\sigma} \;\sqsubseteq\; \lambda z : \sigma\times\tau . ?_{\tau\times\sigma} \;\sqsubseteq\; \lambda z : \sigma\times\tau . \langle \pi_2 z, \pi_1 z \rangle$$

Formally, this is

$$\dfrac{\dfrac{}{\vdash ?_{\sigma\times\tau\to\tau\times\sigma} \sqsubseteq \lambda z : \sigma\times\tau . ?_{\tau\times\sigma}}\,\text{ABS-REF} \qquad \dfrac{}{\vdash \lambda z : \sigma\times\tau . ?_{\tau\times\sigma} \sqsubseteq \lambda z : \sigma\times\tau . \langle \pi_2 z, \pi_1 z \rangle}}{\vdash ?_{\sigma\times\tau\to\tau\times\sigma} \sqsubseteq \lambda z : \sigma\times\tau . \langle \pi_2 z, \pi_1 z \rangle}\,\text{TRANS-REF}$$

The first hypothesis follows from (ABS-REF), and the second is derived as

$$\dfrac{\dfrac{\dfrac{}{z : \sigma\times\tau \vdash ?_{\tau\times\sigma} \sqsubseteq \langle ?_\tau, ?_\sigma \rangle}\,\text{PAIR-REF} \qquad \dfrac{\dfrac{\dfrac{}{y : \tau \vdash ?_\tau \sqsubseteq y}\,\text{VAR-REF}}{z : \sigma\times\tau \vdash ?_\tau \sqsubseteq \pi_2 z}\,\text{PROJ2-REF} \quad \dfrac{\dfrac{}{x : \sigma \vdash ?_\sigma \sqsubseteq x}\,\text{VAR-REF}}{z : \sigma\times\tau \vdash ?_\sigma \sqsubseteq \pi_1 z}\,\text{PROJ1-REF}}{z : \sigma\times\tau \vdash \langle ?_\tau, ?_\sigma \rangle \sqsubseteq \langle \pi_2 z, \pi_1 z \rangle}\,\text{PAIR-}\xi\text{-REF}}{z : \sigma\times\tau \vdash ?_{\tau\times\sigma} \sqsubseteq \langle \pi_2 z, \pi_1 z \rangle}\,\text{TRANS-REF}}{\vdash \lambda z : \sigma\times\tau . ?_{\tau\times\sigma} \sqsubseteq \lambda z : \sigma\times\tau . \langle \pi_2 z, \pi_1 z \rangle}\,\text{ABS-}\xi\text{-REF}$$

4 Models

We can interpret the calculus using a simple generalisation of Henkin models [3]. A Henkin model (with products) is an applicative structure, that is, a tuple of families of sets and type-indexed mappings

$$\langle \{A^\sigma\}, \{\texttt{Proj}_1^{\sigma,\tau}\}, \{\texttt{Proj}_2^{\sigma,\tau}\}, \{\texttt{App}^{\sigma,\tau}\}, \texttt{Const} \rangle$$

which is also extensional and satisfies the environment model condition. See [6] for details. It is extensionality which allows us to interpret abstractions, pairs and the unit uniquely, and the environment model condition which gives us enough elements. To each type σ we ascribe the set A^σ, and use `Const` to map constants into the appropriate sets. This much is needed for a model of the simply-typed lambda calculus. We will use the same apparatus, but interpret our underdetermined terms as subsets of the A^σ, rather than elements[2].

We make one additional assumption of our models. We require that the function sets be 'closed under factoring'. Let $f' \in A^{\tau\to\tau'}$, $f \in A^{\sigma\to\tau'}$. If for all $b \in A^\tau$, there exists an $a \in A^\sigma$ such that $\texttt{App}(f', b) = \texttt{App}(f, a)$, then there exists an element $g \in A^{\tau\to\sigma}$ such that for all $b \in A^\tau$, $\texttt{App}(f', b) = \texttt{App}(f, \texttt{App}(g, b))$. In other words, writing $\bar{f}$ for the function associated with $f \in A^{\sigma\to\tau}$, that is, $(a \in A^\sigma \mapsto \texttt{App}(f, a))$: if there exists a function $h : A^\tau \to A^\sigma$ such that $\bar{f}' = h; \bar{f}$, then there exists an element $g \in A^{\tau\to\sigma}$ such that $\bar{f}' = \bar{g}; \bar{f}$. Note that we do not require $\bar{g} = h$. The factoring condition is essentially a form of choice axiom.

Before proceeding further, we give an example of a Henkin model with the factoring condition.

[2]We need to interpret $\Gamma \vdash r : \tau$ at $\Gamma \to \mathcal{P}(A^\tau)$ rather than $\mathcal{P}(A^\Gamma \to A^\tau)$, to capture the fact that underdeterminism depends on the context. For example, there are only two elements in $\mathcal{P}(A^{\mathbf{1}} \to A^{\mathbf{1}})$, but in context $b : \texttt{bool}$, each of $*$, 0 and $\texttt{if}\ b\ \texttt{then}\ *\ \texttt{else}\ \texttt{empty}_1$, where $\texttt{empty}_1$ is an empty term of type $\mathbf{1}$, should have different interpretations.

Example 4.1 (Full set-theoretic function hierarchy) The sets of the full set-theoretic function hierarchy for a given signature are defined inductively as

$$A^{\gamma} = \text{ any set}$$

$$A^{\mathbf{1}} = \{*\}$$

$$A^{\sigma\times\tau} = A^{\sigma} \times A^{\tau}$$

$$A^{\sigma\to\tau} = A^{\sigma} \to A^{\tau}$$

The projection and application maps are the usual set-theoretic maps, and `Const` is any mapping into the appropriate sets.

We will see below that the open term model is another example. Now we interpret $\Gamma \vdash r : \tau$ inductively on the structure of the typing derivation. This is given in Figure 6.

$$[\![\Gamma, x:\sigma, \Gamma' \vdash x : \sigma]\!]\langle g, a, g'\rangle = \{a\}$$

$$\frac{[\![\Gamma \vdash r_i]\!] = m_i}{[\![\Gamma \vdash k(r_1, \ldots, r_n) : \tau]\!](g) = \{\mathtt{Const}(k)(a_1, \ldots a_n) \mid a_i \in m_i(g)\}}$$

$$[\![\Gamma \vdash * : \mathbf{1}]\!](g) = A^{\mathbf{1}}$$

$$\frac{[\![\Gamma \vdash r : \sigma]\!] = m \quad [\![\Gamma \vdash r' : \tau]\!] = m'}{[\![\Gamma \vdash \langle r, r'\rangle : \sigma\times\tau]\!](g) = \{a \in A^{\sigma\times\tau} \mid \mathtt{Proj}_1^{\sigma,\tau}(a) \in mg, \mathtt{Proj}_2^{\sigma,\tau}(a) \in m'g\}}$$

$$\frac{[\![\Gamma, x : \sigma \vdash r : \tau]\!] = m}{[\![\Gamma \vdash \lambda x:\sigma.r : \sigma\to\tau]\!](g) = \{f \in A^{\sigma\to\tau} \mid \forall a \in A^{\sigma}.\mathtt{App}(f, a) \in m\langle g, a\rangle\}}$$

$$[\![\Gamma \vdash ?_{\sigma} : \sigma]\!](g) = A^{\sigma}$$

$$\frac{[\![\Gamma \vdash r : \tau\times\tau']\!] = m}{[\![\Gamma \vdash \pi_1(r) : \tau]\!](g) = \{\mathtt{Proj}_1^{\tau,\tau'}(a) \mid a \in mg\}}$$

$$\frac{[\![\Gamma \vdash r : \tau\times\tau']\!] = m}{[\![\Gamma \vdash \pi_2(r) : \tau]\!](g) = \{\mathtt{Proj}_2^{\tau,\tau'}(a) \mid a \in mg\}}$$

$$\frac{[\![\Gamma \vdash r : \sigma\to\tau]\!] = m \quad [\![\Gamma \vdash r' : \sigma]\!] = m'}{[\![\Gamma \vdash rr' : \tau]\!](g) = \{\mathtt{App}(f, a) \mid f \in mg, a \in m'g\}}$$

$$\frac{[\![\Gamma \vdash r : \sigma]\!] = m \quad [\![\Gamma, x:\sigma \vdash r' : \tau]\!] = m'}{[\![\Gamma \vdash \mathtt{let}\; x:\sigma\; \mathtt{be}\; r\; \mathtt{in}\; r' : \tau]\!](g) = \bigcup_{a\in mg} m'\langle g, a\rangle}$$

Figure 6: Interpretation of well-formed terms

We take an environment g for context Γ to be a tuple of elements in the appropriate sets, written $g \vDash \Gamma$. It is easily seen that we have soundness of typing, that is, if $g \vDash \Gamma$, then $[\![\Gamma \vdash r : \tau]\!](g) \subseteq A^{\tau}$.

We say that a refinement is valid, $\Gamma \vDash r \sqsubseteq_{\tau} r'$, when for all Henkin models which satisfy the equations in the signature, for all $g \vDash \Gamma$, $[\![\Gamma \vdash r : \tau]\!](g) \supseteq [\![\Gamma \vdash r' : \tau]\!](g)$.

We will prove soundness below, but first, it is easy to see that for determined t, the interpretation $[\![\Gamma \vdash t : \tau]\!](g)$ is a singleton set. Now we prove a standard lemma.

Lemma 4.1 *(Substitution Lemma) For well-formed terms in context* $x_1 : \tau_1, \ldots x_n : \tau_n \vdash r : \tau$ *and* $\Gamma \vdash t_i : \tau_i$ $(i = 1, \ldots, n)$, *we have*

$$[\![x_1 : \tau_1, \ldots x_n : \tau_n \vdash r : \tau]\!]\langle a_1, \ldots, a_n\rangle = [\![\Gamma \vdash r[t_i/x_i]]\!](g)$$

where a_i *is the unique inhabitant of* $[\![\Gamma \vdash t_i : \tau_i]\!](g)$.

Proof: The proof is a straightforward induction over the typing judgement $x_1 : \tau_1, \ldots x_n : \tau_n \vdash r : \tau$. ■

Theorem 4.1 *(Soundness) If* $\Gamma \vdash r \sqsubseteq_\tau r'$ *then* $\Gamma \vDash r \sqsubseteq_\tau r'$.

Proof: The proof is by induction on the derivation of the judgement. Most cases are straightforward. We prove two key cases.

- (LET-β-EQ)
  The interpretation $[\![\Gamma \vdash \mathtt{let}\ x : \tau\ \mathtt{be}\ t\ \mathtt{in}\ r : \tau']\!](g)$ is defined to be $\bigcup_{a \in [\![\Gamma \vdash t:\tau]\!](g)} [\![\Gamma, x : \tau \vdash r]\!]\langle g, a\rangle$. Since t is determined, this is $[\![\Gamma, x : \tau \vdash r : \tau']\!]\langle g, a\rangle$ where a is the unique member of $[\![\Gamma \vdash t : \tau]\!](g)$, so by the substitution lemma, the interpretation is $[\![\Gamma \vdash r[t/x] : \tau']\!](g)$.

- (LET-ABS-EQ).
  To prove the soundness of LET-ABS-EQ, we show first that the axiom follows from the simpler

$$\frac{\Gamma, x : \sigma, y : \tau \vdash t[x, y] : \tau'}{\Gamma \vdash \mathtt{let}\ z : \sigma \to \tau\ \mathtt{in}\ \lambda x : \sigma.t[x, zx] =_{\sigma \to \tau'} \lambda x : \sigma.(\mathtt{let}\ y : \tau\ \mathtt{in}\ t[x, y])} \tag{5}$$

where the term t is determined. The first step is to show, by induction over terms, that (5) is sufficient to prove that all terms have a canonical form. For the abstraction case, we have $\lambda x : \sigma.r = \lambda x : \sigma.\mathtt{let}\ y : \tau\ \mathtt{in}\ t[x, y]$, by induction, and using (5) we get $\mathtt{let}\ f : \sigma \to \tau\ \mathtt{in}\ \lambda x : \sigma.t[x, fx]$. Now we prove the full axiom, (LET-ABS-EQ). Assume $\lambda x : \sigma.\mathtt{let}\ y : \tau\ \mathtt{in}\ r\ :\ \sigma \to \tau''$.

$$\begin{array}{rcl}
\lambda x : \sigma.\mathtt{let}\ y : \tau\ \mathtt{in}\ r[x, y] & = & \lambda x : \sigma.\mathtt{let}\ y : \tau, z : \tau'\ \mathtt{in}\ t[y, z] \\
& = & \lambda x : \sigma.\mathtt{let}\ p : \tau \times \tau'\ \mathtt{in}\ t[\pi_1 p, \pi_2 p] \\
& = & \mathtt{let}\ f'' : \sigma \to \tau \times \tau'\ \mathtt{in}\ \lambda x : \sigma.t[\pi_1(f''x), \pi_2(f''x)] \\
& = & \mathtt{let}\ f : \sigma \to \tau, f' : \sigma \to \tau'\ \mathtt{in}\ \lambda x : \sigma.t[fx, f'x] \\
& = & \mathtt{let}\ f : \sigma \to \tau\ \mathtt{in}\ \lambda x : \sigma.\mathtt{let}\ z : \tau'\ \mathtt{in}\ t[fx, z] \\
& = & \mathtt{let}\ f : \sigma \to \tau\ \mathtt{in}\ \lambda x : \sigma.r[x, fx]
\end{array}$$

Now we prove that (5) is sound.

Let $g \vDash \Gamma$. Now $f \in [\![\Gamma \vdash \mathtt{let}\ z : \sigma \to \tau\ \mathtt{be}\ ?_{\sigma \to \tau}\ \mathtt{in}\ \lambda x : \sigma.t[x, zx] : \sigma \to \tau']\!](g)$, that is, $\bigcup_{a \in A^{\sigma \to \tau}} [\![\Gamma, z : \sigma \to \tau \vdash \lambda x : \sigma.t[x, zx] : \sigma \to \tau']\!]\langle g, a\rangle$, when

$$\exists a \in A^{\sigma \to \tau} \text{ such that } \forall b \in A^\sigma, fb \in [\![\Gamma, z : \sigma \to \tau, x : \sigma \vdash t[x, zx] : \tau']\!]\langle g, a, b\rangle \tag{6}$$

and $f \in [\![\Gamma \vdash \lambda x : \sigma.(\mathtt{let}\ y : \tau\ \mathtt{be}\ ?_\tau\ \mathtt{in}\ t[x, y]) : \sigma \to \tau']\!](g)$, when

$$\forall b \in A^\sigma, \exists a_b \in A^\tau \text{ such that } fb \in [\![\Gamma, x : \sigma, y : \tau \vdash t[x, y] : \tau']\!]\langle g, b, a_b\rangle \tag{7}$$

We must prove (6) and (7) are equivalent. Suppose (6). Now let $b \in A^\sigma$. We can define a_b as $\mathtt{App}(a, b)$. We have $fb \in [\![\Gamma, z : \sigma \to \tau, x : \sigma \vdash t[x, zx] : \tau']\!]\langle g, a, b\rangle = [\![\Gamma, x : \sigma, y : \tau \vdash t[x, y] : \tau']\!]\langle g, b, a_b\rangle$, so (7) holds.

Now suppose (7). Define $a' : A^\sigma \to A^\tau$ as $(b \in A^\sigma \mapsto a_b \in A^\tau)$, and $h : A^\sigma \to A^{\sigma \times \tau}$ to be $(b \in A^\sigma \mapsto \langle b, a(b)\rangle)$. Then (7) can be read as $h; \bar{f}' = \bar{f}$ where $f' : A^{\sigma \times \tau \to \tau'}$ is the unique inhabitant of $[\![\Gamma \vdash \lambda p : \sigma \times \tau.t[\pi_1 p, \pi_2 p] : \sigma \times \tau \to \tau']\!](g)$.

By the factoring condition, there exists an element $g \in A^{\sigma \to \sigma \times \tau}$ such that $\bar{g}; \bar{f}' = \bar{h}$. Now define $a = \mathtt{App}(\mathtt{App}(comp, g), p_2) \in A^{\sigma \to \tau}$, where the elements $comp = [\![\lambda f : \sigma \to \sigma \times \tau.\lambda g : \sigma \times \tau \to$

$\tau.\lambda x:\sigma.g(fx)]\!]$ and $p_2 = [\![\lambda p:\sigma\times\tau.\pi_2(p)]\!]$ exist by the environment model condition, and so $a\in A^{\sigma\to\tau}$.

We now prove (6). Let $b\in A^\sigma$. Then, by assumption, $fb\in[\![\Gamma,x:\sigma,y:\tau\vdash t[x,y]:\tau']\!]\langle g,b,a_b\rangle$ which, by the substitution lemma, is $[\![\Gamma,z:\sigma\to\tau,x:\sigma\vdash t[x,zx]:\tau']\!]\langle g,a,b\rangle$, so (6) holds.

∎

Since we have nontrivial models, soundness implies that the calculus is consistent. More significantly, however, we have completeness of the equational theory of simply-typed underdeterminism with respect to the class of Henkin models with factoring. This implies the system is a conservative extension of the simply-typed lambda calculus.

We will need the following lemmas. The first says that all auxiliary claims made using `let` weakening can be immediately satisfied.

Lemma 4.2 *If* $\Gamma\vdash \texttt{let}\ z:\sigma\ \texttt{be}\ r\ \texttt{in}\ r'\sqsubseteq_\tau t'$ *then there exists a determined term* $\Gamma\vdash t:\sigma$ *such that* $\Gamma\vdash r\sqsubseteq_\sigma t$ *and* $\Gamma\vdash r'[t/z]\sqsubseteq_\tau t'$.

The lemma follows from

Lemma 4.3 *Suppose* $\Gamma\vdash r\sqsubseteq_\tau t'$*, and* r *has the form* $t[?^{\Gamma_1}_{\sigma_1},\ldots,?^{\Gamma_n}_{\sigma_n}]$*, where* $?^{\Gamma_i}_{\sigma_i}$ *means that the subterm* $?_{\sigma_i}$ *appears in 'local context' (of* `let`*'s and* λ*'s)* Γ_i*. Note also that each such subterm appears exactly once.*

Then for each $i=1\ldots n$*, there exists* $\Gamma,\Gamma_i\vdash t_i:\sigma_i$*, such that* $\Gamma\vdash t[t_1,\ldots,t_n]=_\tau t'$.

Proof: We can prove the lemma by induction on the derivation of refinement. Write $\Gamma\vdash r\leadsto t'$ if there exists a standard refinement sequence given by immediately refining the specifications in $r\equiv t[?^{\Gamma_1}_{\sigma_1},\ldots,?^{\Gamma_n}_{\sigma_n}]$ as $\Gamma,\Gamma_i\vdash ?_{\sigma_i}\sqsubseteq u_i$. The inductive case involves showing that the standard sequence can be extended back along each rule. We consider two cases. The reasoning is similar for the others.

- (LET-WEAKEN-REF)
  Suppose $\Gamma\vdash r\sqsubseteq \texttt{let}\ x:\sigma\ \texttt{be}\ r'\ \texttt{in}\ r\leadsto t''$, where $r\equiv t_1[?^{\Gamma_1}_{\sigma_1},\ldots,?^{\Gamma_n}_{\sigma_n}]$. By the inductive hypothesis, there exists a t' and $t_2[x]$ such that $\Gamma\vdash r'\sqsubseteq t'$ (the details of this refinement do not matter), and $\Gamma,x:\sigma\vdash r\leadsto t_2[x]$ with $\Gamma\vdash \texttt{let}\ x:\sigma\ \texttt{be}\ t'\ \texttt{in}\ t_2=t''$, so $\Gamma\vdash t_2[t']=t''$. Now the standard refinement of r gives terms $\Gamma,x:\sigma,\Gamma_i\vdash u_i:\sigma_i$ for each $i=1\ldots n$, and so $\Gamma,\Gamma_i\vdash u_i[t'/x]:\sigma_i$.

  Hence, $\Gamma\vdash r\leadsto t_2[t']=t''$, by refining $?^{\Gamma_i}_{\sigma_i}$ to $u_i[t_i/x]$.

- (LET-ABS-EQ)
  Since this rule is given as an equation, we must consider the two directions of refinement separately. Suppose

  $$\Gamma\vdash \texttt{let}\ z:\sigma\to\tau\ \texttt{in}\ \lambda x:\sigma.t'[?^{\Gamma_i}_{\sigma_i},x,zx]\ \sqsubseteq\ \lambda x:\sigma.(\texttt{let}\ y:\tau\ \texttt{in}\ t'[?^{\Gamma_i}_{\sigma_i},x,y])\leadsto t$$

  For clarity, we just indicate the one specification $?_{\sigma_i}$. Here Γ_i records the context in t'.

  By induction, there exist terms $\Gamma,x:\sigma,y:\tau,\Gamma_i\vdash u_i[x,y]:\sigma_i$ and $\Gamma,x:\sigma\vdash u[x]:\tau$ such that $\Gamma\vdash\lambda x:\sigma.(\texttt{let}\ y:\tau\ \texttt{be}\ u\ \texttt{in}\ t'[u_i,x,y]=t$, that is, $\Gamma\vdash\lambda x:\sigma.t'[u_i[x,u],x,u]=t$.

  Then we have $\Gamma\vdash\lambda x:\sigma.u[x]:\sigma\to\tau$ and $\Gamma,z:\sigma\to\tau,x:\sigma,\Gamma_i\vdash u_i[x,zx]:\sigma_i$ so we can refine the left term to $\Gamma\vdash \texttt{let}\ z:\sigma\to\tau\ \texttt{be}\ \lambda x:\sigma.u[x]\ \texttt{in}\ \lambda x:\sigma.t'[u_i[x,zx],x,zx]$, which equals $\lambda x:\sigma.t'[u_i[x,u[x]],x,u[x]]$, and this equals t.

  Now we consider the reverse refinement. Suppose

  $$\Gamma\vdash\lambda x:\sigma.(\texttt{let}\ y:\tau\ \texttt{in}\ t'[?^{\Gamma_i}_{\sigma_i},x,y])\ \sqsubseteq\ \texttt{let}\ z:\sigma\to\tau\ \texttt{in}\ \lambda x:\sigma.t'[?^{\Gamma_i}_{\sigma_i},x,zx]\leadsto t$$

By induction, we have terms $\Gamma \vdash u : \sigma \to \tau$ and $\Gamma, z : \sigma \to \tau, x : \sigma, \Gamma_i \vdash u_i[z, x] : \sigma_i$ such that

$$\Gamma \vdash \texttt{let}\ z : \sigma \to \tau\ \texttt{be}\ u\ \texttt{in}\ \lambda x : \sigma.t'[u_i[z, x], x, zx] = t$$

so $\Gamma \vdash \lambda x : \sigma.t'[u_i[u, x], x, ux] = t$.

Hence we have terms

$$\Gamma, x : \sigma \vdash ux : \tau$$

$$\Gamma, x : \sigma, y : \tau, \Gamma_i \vdash u_i[u, x] : \sigma_i$$

for which

$$\Gamma \vdash \lambda x : \sigma.(\texttt{let}\ y : \tau\ \texttt{be}\ ux\ \texttt{in}\ t'[u_i[u, x], x, y]) = t$$

and so we can refine the left term in a standard sequence.

∎

This lemma tells us that if a refinement is possible, then there exists a 'standard refinement sequence' of this form.

Remark 4.1 Note that this lemma only holds if the axioms of applied theories only involve determined equations, $t =_\tau t'$. If a theory has constants $k_1, k_2, k_3 : \tau$ and axiom $\texttt{let}\ b : \texttt{bool}\ \texttt{in}\ (\texttt{if}\ b\ \texttt{then}\ k_1\ \texttt{else}\ k_2) \sqsubseteq_\tau k_3$, then we cannot find a specific $t : \texttt{bool}$ such that $\texttt{if}\ t\ \texttt{then}\ k_1\ \texttt{else}\ k_2 =_\tau k_3$.

Theorem 4.2 *(Completeness of equational system) If all types are inhabited, then* $\Gamma \vdash r \sqsubseteq_\tau r'$ *iff* $\Gamma \models r \sqsubseteq_\tau r'$

Proof : We give a sketch of the proof. The idea is to construct a minimal term model for our signature of ground types, constants and equational assumptions (with no empty types).

1. Fix an infinite context Γ_∞ with an infinite number of variables at each type. We use Γ_∞ in judgements to mean some finite $\Gamma' \subseteq \Gamma_\infty$.

2. Define A^τ as the set of $=_\tau$-equivalence classes of open determined terms of type τ, that is, $A^\tau = \{[t]^\tau \mid \Gamma_\infty \vdash t : \tau\}$. The projection, application and constant interpretation mappings are interpreted syntactically. Since provable equality is a logical equivalence relation, this forms a Henkin model [6]. A^τ is nonempty since we have variables at each type.

   To see that the model satisfies the factoring condition, suppose $h; \bar{f} = \bar{f}'$, where $f = [x : \sigma \vdash t[x] : \tau']$ and $f' = [y : \tau \vdash t'[y] : \tau']$. This says that for all terms $u' : \tau$ there exists a term $h(u') : \sigma$ such that $t[h(u')] = t'[u']$. In particular then, for the variable $y : \tau$, there exists some term $u[y] : \sigma$ such that $t[u[y]] = t'[y]$. We can define g to be $\lambda y : \tau.u[y]$, and then $\bar{g}; \bar{f} = \bar{f}'$.

3. Prove that $[\![\Gamma \vdash r : \tau]\!](g) = \{[t]^\tau \mid \Gamma_\infty \vdash r[g/\Gamma] \sqsubseteq_\tau t\}$, where $r[g/\Gamma]$ has the obvious meaning. The $?_\sigma$ case uses Lemma 3.3. We prove the $\texttt{let}\ z : \sigma\ \texttt{be}\ r\ \texttt{in}\ r'$ case.

   The interpretation is

   $$\begin{array}{rcl} [\![\Gamma \vdash \texttt{let}\ z : \sigma\ \texttt{be}\ r\ \texttt{in}\ r']\!](g) & = & \bigcup_{a \in [\![\Gamma \vdash r:\sigma]\!](g)} [\![\Gamma, z : \sigma \vdash r']\!]\langle g, a\rangle \\ & = & \bigcup_{a \in \{[t] \mid r[g/\Gamma] \sqsubseteq t\}} \{[t'] \mid r'[g/\Gamma, a/z] \sqsubseteq t'\} \end{array}$$

   Now t' is in the set when there exists a t such that $r[g/\Gamma] \sqsubseteq t$ and $r'[g/\Gamma, t/z] \sqsubseteq t'$. Hence $(\texttt{let}\ z : \sigma\ \texttt{be}\ r\ \texttt{in}\ r')[g/\Gamma] \sqsubseteq t'$.

   Conversely, if $\texttt{let}\ z : \sigma\ \texttt{be}\ r[g/\Gamma]\ \texttt{in}\ r'[g/\Gamma] \sqsubseteq t'$, then by Lemma 4.2, there exists a t such that $r[g/\Gamma] \sqsubseteq t$ and $r'[g/\Gamma, t/z] \sqsubseteq t'$.

4. Prove that $\Gamma \models r \sqsubseteq_\tau r'$ iff $\Gamma \vdash r \sqsubseteq_\tau r'$.

The first three steps are standard in completeness proofs; the fourth is particular to our calculus. To prove it, we first define an ordering $r \lesssim_\tau r'$, for well-formed open terms $r : \tau$, $r' : \tau$, to mean for all determined open terms $t : \tau$, $\Gamma_\infty \vdash r' \sqsubseteq_\tau t \Rightarrow \Gamma_\infty \vdash r \sqsubseteq_\tau t$. Note that $\Gamma \vDash r \sqsubseteq_\tau r'$, is equivalent to $r \lesssim_\tau r'$. Clearly $r \sqsubseteq_\tau r' \Rightarrow r \lesssim_\tau r'$. To get completeness, we need to prove the converse. The crucial step uses the canonical form of refinement terms :

Lemma 4.4 *(Refinement mappings) If*

$$\mathtt{let}\ x_1 : \sigma_1, \ldots, x_n : \sigma_n\ \mathtt{in}\ t \quad \lesssim_\tau \quad \mathtt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \mathtt{in}\ t'$$

then letting $\Gamma \equiv y_1 : \tau_1, \ldots, y_m : \tau_m$,
for all $i = 1 \ldots n$, *there exists* $\Gamma \vdash t_i : \sigma_i$, *such that* $\Gamma \vdash t[t_1/x_1, \ldots, t_n/x_n] =_\tau t'$.

Proof: First of all, note that this can be reduced by repeated pairing to the one variable case. Now if $\mathtt{let}\ x : \sigma\ \mathtt{in}\ t \lesssim_v \mathtt{let}\ y : \tau\ \mathtt{in}\ t'$, then since $\mathtt{let}\ y : \tau\ \mathtt{in}\ t' \sqsubseteq t'[y]$, we have $\mathtt{let}\ x : \sigma\ \mathtt{in}\ t \sqsubseteq t'[y]$, so by Lemma 4.2, there must exist a term $u[y]$ such that $t[u[y]] =_v t'$. ∎

Now we can finish the proof of completeness. Suppose $r \lesssim_\tau r'$. We can express this using canonical forms as

$$\mathtt{let}\ x_1 : \sigma, \ldots, x_n : \sigma_n\ \mathtt{in}\ t \quad \lesssim_\tau \quad \mathtt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \mathtt{in}\ t'$$

so assume $r \equiv \mathtt{let}\ x_1 : \sigma, \ldots, x_n : \sigma_n\ \mathtt{in}\ t$ and $r' \equiv \mathtt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \mathtt{in}\ t$. By LET-WEAKEN-REF, $r \sqsubseteq_\tau \mathtt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \mathtt{in}\ r$ and this refines to $\mathtt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \mathtt{in}\ t[t_1/x_1, \ldots, t_n/x_n]$ using Lemma 4.4. Again by Lemma 4.4, this is equal to $\mathtt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \mathtt{in}\ t'$ which equals r'. Hence $\Gamma_\infty \vdash r \sqsubseteq_\tau r'$, and since types are nonempty we get $\Gamma \vdash r \sqsubseteq_\tau r'$. ∎

Corollary 4.1 *If* $\Gamma \vdash r : \sigma$ *then* $\Gamma \vdash ?_\sigma \sqsubseteq_\sigma r$.

Corollary 4.2 $\lambda_?$ *is a conservative extension of* $\lambda^{\times\to}$.

Proof: Both calculi are complete for Henkin models (with the factoring condition), and the determined equation $\Gamma \vdash t =_\tau t'$ has the same interpretation. ∎

5 Logic

We can consider an auxiliary logic built on top of the equational theory, with judgements

$$\Delta \vdash_\Gamma P$$

where Δ is a list of propositions, and P a proposition. The meaning is : for all variables in Γ, if each proposition in Δ is true, then P is true.

This combination of logic and refinement is not the same as was mentioned in the brief discussion at the end of Section 1. The idea discussed there is to internalise logic into the stubs themselves. In this section we present an 'external' logic for reasoning about refinement.

We should regard the logic as being orthogonal to refinement. Here we use classical first order logic with refinements $r \sqsubseteq_\tau r'$ as atomic propositions. The propositions of the logic are

$$P ::= \mathtt{false} \mid P \supset P' \mid \forall x : \tau . P \mid r \sqsubseteq_\tau r'$$

These are sufficient to define $\wedge$, $\vee$, $\exists$ and $\neg$. In addition to the rules for refinement, we assume a signature of (extralogical) axioms, which we can take, without loss of generality, to be closed propositions. We write $Ax \triangleright P$ when P is in this collection. We only allow equations between determined terms, $t =_\tau t'$ (that is, $t \sqsubseteq_\tau t' \wedge t' \sqsubseteq_\tau t$) as axioms. This is a natural restriction, given our intuition of constructing a refinement theory on top of an applied $\lambda^{\times\to}$ theory.

The natural deduction rules for the judgement $\Delta \vdash_\Gamma P$ of first order logic are given in the appendix. Only the rules for $\supset$, $\forall$ and `false` are necessary, but we give the derived rules for the other connectives as well. We must also add a rule

$$\frac{\Gamma \vdash r \sqsubseteq_\tau r'}{\Delta \vdash_\Gamma r \sqsubseteq_\tau r'} \text{ (REF)}$$

so that the equational theory is subsumed by the logic.

There are also well-formedness judgements

$$\Gamma \vdash P \ \texttt{wf}$$

$$\Gamma \vdash \Delta \ \texttt{wf}$$

which we omit here.

The interest in studying the logic is that it is complete over the same class of models as the equational theory. After giving a semantics and proving completeness below, we will be able to conclude that refinement can be encoded in the logic using just equality. This does not mean that the notion of refinement is superfluous. Rather, we can consider $r \sqsubseteq_\tau r'$ to be a useful high-level notation for some proposition with quantifiers.

In Figure 7 we give the interpretation for the propositions. We interpret a proposition in context

$$[\![\Gamma \vdash \texttt{false}]\!] = \emptyset$$
$$[\![\Gamma \vdash P \supset P']\!] = \{g \vDash \Gamma \ . \ g \notin [\![\Gamma \vdash P]\!] \text{ or } g \in [\![\Gamma \vdash P']\!]$$
$$[\![\Gamma \vdash \forall x : \sigma . P]\!] = \{g \vDash \Gamma \mid \forall a \in A^\sigma . \langle g, a\rangle \in [\![\Gamma, x : \sigma \vdash P]\!]\}$$
$$[\![\Gamma \vdash r \sqsubseteq_\sigma r']\!] = \{g \vDash \Gamma \mid [\![\Gamma \vdash r : \sigma]\!](g) \supseteq [\![\Gamma \vdash r' : \sigma]\!](g)\}$$

Figure 7: Interpretation of well-formed propositions

$\Gamma \vdash P$ as the set of environments $g \vDash \Gamma$ in which P holds. We say that $\Delta \vDash_\Gamma P$ in some model, if for all $g \vDash \Gamma$, if $g \vDash A$ for each A in Δ, then $g \vDash P$. Completeness is with respect to the class of Henkin models (of the applied theory) with the factoring condition.

Theorem 5.1 *(Soundness and Completeness of logical system)* $\Delta \vdash_\Gamma P$ *iff* $\Delta \vDash_\Gamma P$

Proof: Soundness is straightforward to prove. As for completeness, we do not have a minimal (Henkin) model for the logical system, but nevertheless, we can still use a term model in order to prove deductive completeness, by showing that any consistent theory is satisfiable. For a theory with axioms Ax, we want to show that $\Delta \vdash_\Gamma P$ iff $\Delta \vDash_\Gamma P$ in all Henkin models of Ax with the factoring condition. We can restrict our attention to closed propositions. We do not assume that types are nonempty.

1. Given $Ax, \Delta \nvdash P$ we want to find a Henkin model $\mathcal{A}$ such that $\mathcal{A} \vDash Ax \cup \Delta \cup \{\neg P\}$.

2. Construct a maximal consistent Henkin theory Δ_∞ such that $Ax \cup \Delta \cup \{\neg P\} \subseteq \Delta_\infty$.

   A *Henkin*[3] *theory*, T, is a collection of propositions closed under derivation, such that if the proposition $\exists x : \tau . P$ is in T, then $P[t/x]$ is in T for some $t : \tau$. See [10].

   Here we extend the definition of Henkin theory with the further condition that if the proposition $\texttt{let}\ x : \tau\ \texttt{in}\ t \sqsubseteq_\sigma \texttt{let}\ x' : \tau'\ \texttt{in}\ t'$ is in T, then there exists a term $t'' : \tau' \to \tau$ such that $\forall x' : \tau' . t[t''x'/x] =_\sigma t'$ is in T.

   The construction of the Henkin theory involves adding constants $k_P : \tau$, and axioms $P[k_P]$, for each proposition P, for which $\exists x : \tau . P$ is in T, and constants $f : \tau' \to \tau$ and the corresponding axiom, $\forall x' : \tau' . t[fx'/x] =_\sigma t'$, for every $\texttt{let}\ x : \tau\ \texttt{in}\ t \sqsubseteq_\sigma \texttt{let}\ x' : \tau'\ \texttt{in}\ t'$ in T.

[3]Nothing to do with Henkin models.

The existence of a maximal consistent Henkin extension of $Ax \cup \Delta \cup \{\neg P\}$ follows using a standard argument from Zorn's Lemma. See [10] for details. Note that for a maximal consistent theory T and well-formed proposition P, exactly one of P, $\neg P$ is in T.

3. Define an infinite context Γ_∞ consisting of an infinite number of variables $x_i : \sigma$ for each type σ for which the proposition $\exists x : \sigma.\mathtt{true}$ is in Δ_∞.

4. Define a family of relations on open determined terms $R^\sigma_{\Delta_\infty}(t, t') \iff \Delta_\infty \vdash_{\Gamma_\infty} t =_\sigma t'$ and prove it is a logical equivalence relation. See [6] for details.

5. Construct the term model $\mathcal{T}$ of the calculus, and prove that $[\![\Gamma \vdash r : \tau]\!]_\mathcal{T}(g) = \{[t]^\tau \mid \Gamma_\infty \vdash r[g/\Gamma] \sqsubseteq_\tau t\}$.

6. Construct the quotient model $\mathcal{A} = \mathcal{T}/R$ where A^τ is the set of $R^\tau_{\Delta_\infty}$-classes, and show that $[\![\Gamma \vdash r : \tau]\!]_\mathcal{A}(g) = \{[t]^\tau_R \mid \Delta_\infty \vdash_{\Gamma_\infty} r[g/\Gamma] \sqsubseteq_\tau t\}$. This requires the generalisation of Lemma 4.3 to logical contexts. In contrast with Remark 4.1, this holds because of the construction of the Henkin theory Δ_∞.

   $\mathcal{A}$ is a Henkin model with the factoring condition.

7. Prove that $\mathcal{A} \models P$ iff $P \in \Delta_\infty$. The crucial cases are $\exists x : \tau.P$ and $r \sqsubseteq_\tau r'$, which go through by virtue of Δ_∞ being a Henkin theory. The $r \sqsubseteq_\sigma r'$ case is proven as in Theorem 4.2, with the observation that the appropriate generalisation of Lemma 4.3 holds. ∎

The use of canonical forms in the proof of Theorem 4.2 suggests that we can translate refinement into first order logic, just using equality of determined terms. If we let r° denote the canonical form of open term r, then if $r^\circ = \mathtt{let}\ x_1 : \sigma_1, \ldots, x_n : \sigma_n\ \mathtt{in}\ t$, and $r'^\circ = \mathtt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \mathtt{in}\ t'$, define $(r \sqsubseteq_v r')^\circ$ to be $\forall y_1 : \tau_1, \ldots, y_m : \tau_m.\exists x_1 : \sigma_1, \ldots, x_n : \sigma_n.t =_v t'$. Then the logical system is a conservative extension of the equational system (which, in turn, is a conservative extension of the simply-typed lambda calculus). Moreover, we have

Corollary 5.1 *If all types are inhabited, then* $\Gamma \vdash r \sqsubseteq_\tau r'$ *iff* $\langle\rangle \vdash_\Gamma r \sqsubseteq_\tau r'$ *iff* $\langle\rangle \vdash_\Gamma (r \sqsubseteq_\tau r')^\circ$.

Proof : Both systems are complete with respect to Henkin models (with the factoring condition), and the statements have the same interpretation. ∎

6 Conclusion

As far as we know, this is the first proof of completeness of any refinement calculus. Though the language we study is very simple, we believe that it captures the essence of program refinement. We plan to extend this to more complicated type theories, the polymorphic lambda calculus offering particular promise as a way of combining data refinement and type theory. We are currently combining the calculus with the calculus of specifications in [2]. This gives a calculus expressive enough to study program development, but still conceptually simple. Another area for development is in restricted calculi more suitable for automation.

We would also be interested to see how this calculus might be usefully combined with work on logical frameworks. The use of logical variables there is an example of underdeterminism.

That the concept of underdeterminism arises both in computing science and in linguistics strengthens our belief that it is an important concept in the study of general informatics.

Acknowledgements

This work has benefited from discussions with my supervisors : Gordon Plotkin, John Power and Marcelo Fiore. The presentation was improved by a number of suggestions from the anonymous referee. Álvaro Moreira made useful comments on drafts.

References

[1] J. Bos. Predicate logic unplugged. In *Tenth Amsterdam Colloquium*, 1995.

[2] E. Denney. Refinement Types for Specification. In *IFIP Working Conference on Programming Concepts and Methods (PROCOMET '98)*, Shelter Island, New York, USA, 1998. To appear.

[3] L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2), 1950.

[4] C. Hermida and B. Jacobs. Fibrations with indeterminates : Contextual and functional completeness for polymorphic lambda calculi. *Mathematical Structures in Computer Science*, 5(4), 1995.

[5] Z. Luo and R. Pollack. LEGO Proof Development System : User's Manual. Technical Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.

[6] J. Mitchell. *Foundations for Programming Languages.* Foundations of Computing Series. MIT Press, 1996.

[7] E. Moggi. Notions of computation and monads. *Information and Computation*, 1, 1991.

[8] C. Morgan. *Programming from Specifications.* Prentice Hall, 1994.

[9] B. Rosser. On the Consistency of Quine's New Foundations for Mathematical Logic. *Journal of Symbolic Logic*, 4, 1939.

[10] D. van Dalen. *Logic and Structure.* Springer-Verlag, 1994.

A Rules for the Logic

$$\Delta \vdash_\Gamma P \qquad (Ax \triangleright P) \qquad \text{(AXIOM)}$$

$$\Delta, P \vdash_\Gamma P \qquad \text{(HYP)}$$

$$\frac{\Delta \vdash_\Gamma P \quad \Delta \vdash_\Gamma Q}{\Delta \vdash_\Gamma P \wedge Q} \qquad (\wedge\text{-INTRO})$$

$$\frac{\Delta \vdash_\Gamma P \wedge Q}{\Delta \vdash_\Gamma P} \qquad (\wedge\text{-ELIM1})$$

$$\frac{\Delta \vdash_\Gamma P \wedge Q}{\Delta \vdash_\Gamma Q} \qquad (\wedge\text{-ELIM2})$$

$$\frac{\Delta \vdash_\Gamma P}{\Delta \vdash_\Gamma \vdash P \vee Q} \qquad (\vee\text{-INTRO1})$$

$$\frac{\Delta \vdash_\Gamma Q}{\Delta \vdash_\Gamma P \vee Q} \qquad (\vee\text{-INTRO2})$$

$$\frac{\Delta \vdash_\Gamma P \vee Q \quad \Delta, P \vdash_\Gamma R \quad \Delta, Q \vdash_\Gamma R}{\Delta \vdash_\Gamma R} \qquad (\vee\text{-ELIM})$$

$$\frac{\Delta, P \vdash_\Gamma Q}{\Delta \vdash_\Gamma P \supset Q} \qquad (\supset\text{-INTRO})$$

$$\frac{\Delta \vdash_\Gamma P \quad \Delta \vdash_\Gamma P \supset Q}{\Delta \vdash_\Gamma Q} \qquad (\supset\text{-ELIM})$$

$$\frac{\Delta \vdash_\Gamma \forall x : \sigma . P \quad \Gamma \vdash t : \sigma}{\Delta \vdash_\Gamma P[t/x]} \qquad (\forall\text{-ELIM})$$

$$\frac{\Delta \vdash_{\Gamma, x:\sigma} P}{\Delta \vdash_\Gamma \forall x : \sigma . P} \; x \notin \Delta \qquad (\forall\text{-INTRO})$$

$$\Delta, \mathtt{false} \vdash_\Gamma P \qquad (\mathtt{false}\text{-ELIM})$$

$$\frac{\Delta, \neg P \vdash_\Gamma \mathtt{false}}{\Delta \vdash_\Gamma P} \qquad (\neg\text{-ELIM})$$

Figure 8: Natural Deduction rules for classical first-order logic