HOOTS99 Preliminary Version

Imprecise Exceptions, Co-Inductively

Andrew Moran

Oregon Graduate Institute
moran@cse.ogi.edu

Søren B. Lassen

University of Cambridge
Soeren.Lassen@cl.cam.ac.uk

Simon Peyton Jones

Microsoft Research Ltd., Cambridge
simonpj@microsoft.com

Abstract

In a recent paper, Peyton Jones *et al.* proposed a design for imprecise exceptions in the lazy functional programming language Haskell [PJRH+99]. The main contribution of the design was that it allowed the language to continue to enjoy its current rich algebra of transformations. However, the denotational semantics used to formalise the design does not combine easily with other extensions, most notably that of concurrency. We present an alternative semantics for a lazy functional language with imprecise exceptions which is entirely operational in nature, and combines well with other extensions, such as I/O and concurrency. The semantics is based upon a convergence relation, which describes evaluation, and an *exceptional* convergence relation, which describes the raising of exceptions. Convergence and exceptional convergence lead naturally to a simple notion of refinement, where a term M is refined by N whenever they have identical convergent behaviour, and any exception raised by N can also be raised by M. We are able to validate many call-by-name equivalences and standard program transformations, including the ubiquitous strictness transformation.

1 Introduction

In an earlier paper [PJRH+99] we showed how to add exceptions to a lazy, purely-functional programming language, such as Haskell. There were three key ideas.

The first was to treat an exception as a *value* rather than as a change of control flow. This idea is fairly standard; for example, the IEEE floating

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

point standard uses it for `NaN`s. The second idea addressed the question of what meaning to assign to expressions like:

$$(\mathsf{raise}\, e_1) + (\mathsf{raise}\, e_2).$$

Does this expression deliver the exception e_1, or e_2? The conventional approach is to fix the evaluation order, thus determining which of the two exceptions is delivered. This works well for languages whose evaluation order is already highly constrained because of other effects, such as assignment or input/output. For languages like Haskell, however, code motion that changes evaluation order is a key transformation, and fixing the evaluation order would be a major blow. The alternative we advocated in [PJRH$^+$99] is to say that the meaning of the expression is a *set* of two exceptions, e_1 and e_2. If the order of the operands to $+$ is reversed, the meaning is unchanged.

The third idea is that to catch an exception is to make a non-deterministic choice among the set of exceptions in an exceptional value. To avoid making the entire language non-deterministic, catching an exception is regarded as an input/output operation in Haskell's I/O monad. This makes our proposal a little less expressive than (say) ML exceptions; the payoff is that program transformations are almost entirely unaffected, with no side conditions.

Our slogan, therefore is: *We want to add exceptions to a lazy language, without losing any useful program transformations.* The earlier paper formalised this claim by using a denotational semantics. In this paper we present an alternative formalisation, by providing an operational semantics for a call-by-name language augmented with raise. There are two reasons for taking this approach:

- Compilers for languages like Haskell take great care to use call by *need*, rather than call by *name*. The two are denotationally indistinguishable, but operationally they may differ dramatically. Despite this, not much theoretical work has been done to make this distinction precise. In separate earlier work we have therefore developed operational techniques to reason about improvement (*i.e.* optimisation) in call-by-need languages [MS99]. We wanted to be able to extend these techniques to a language including exceptions.
- Haskell includes a monadic form of input/output [PJW93], and an extended version of Haskell supports concurrent threads [PJGF96]. We believe that an operational semantics in the style of a process calculus is more suited to describing the semantics of these extensions than is a denotational semantics.

Our goal, then, is to develop a layered operational theory that encompasses input/output, concurrency, exceptions (including asynchronous exceptions such as inter-thread signals), and call-by-need. In the purely functional sublanguage it should be no harder to prove equivalences than it is in simpler calculi; but these equivalences should be proven to hold in the more complex setting in which new kinds of observation are possible.

This paper tackles a small part of that goal. We treat only the purely-functional part, in which one can *raise* an exception, but not *catch* it. We give a formal operational semantics for this language, and prove several equational laws that correspond closely to the standard theorems for a call-by-name calculus [Plo75], thus formalising the effect of adding exceptions on the theory. We study call by name semantics only, leaving the extension to call by need for further work, and we only briefly sketch the extensions to handle I/O and concurrency.

The operational semantics presents a novelty in that exceptional behaviour is defined co-inductively to capture the idea from [PJRH$^+$99] that diverging expressions can raise any exception. Based on this operational semantics we define suitable Morris-style contextual equivalence and refinement relations between terms. As a tool to reason about these, we introduce a form of applicative simulation taking imprecise exceptions into account. We show that the resulting bisimulation equivalence is a congruence, using Howe's congruence proof method [How96], and thus that it coincides with contextual equivalence.

The rest of the paper is organised as follows. Section 2 surveys related work. Section 3 presents the language and its operational semantics in the form of natural semantics rules for *normal convergence* and *exceptional convergence* relations. The next two sections examine two kinds of preorders, based upon the operational semantics. Section 4 presents a Morris-style contextual refinement preorder, where M is refined by N if they have identical convergent behaviour, and any exceptional behaviour exhibited by N is also present in M. We prove that contextual refinement coincides with an applicative simulation preorder, enabling us to establish the validity of beta-laws and the strictness transformation. Extensions to the simple functional language, like case expressions, I/O, exception handling, and concurrency, are then described briefly in section 6. We conclude in section 7.

2 Related Work

The imprecise exception mechanism was introduced in [PJRH$^+$99] where it is related to other approaches to exception handling in functional languages.

Co-inductively defined operational semantics have appeared before (see *e.g.* [CC91,HM95]), but there it has usually been used to define divergence or non-termination predicates. Our use of co-induction to define an exceptional *convergence* relation appears to be new.

There is not much prior work on equational reasoning about exceptions. In [Gor94, §6.7] Gordon considers a small call-by-name language with a (deterministic) monadic exception mechanism, rather different from our imprecise exceptions, for which he also develops an operational theory of applicative bisimulation using Howe's method [How96].

In [PJRH+99] we give a denotational semantics for imprecise exceptions which is carefully engineered to preserve many equational laws from the pure functional language. The operationally-based notion of equivalence relation in the present paper is basically an operational rendering of equality in this denotational model. The refinement preorder that we introduce is inspired by our earlier work on operational theories for non-deterministic functional languages [Las98,Mor98,MSC99]; our adaptation of Howe's congruence proof to deal with non-deterministic exceptions also uses techniques from [Las98].

3 Operational Semantics

In this section, we describe the behaviour of terms in an untyped, call-by-name lambda calculus extended with a data type of exceptions and a means of raising exceptions, called raise. The syntax of the language is:

$$M, N \;::=\; x \mid \lambda x.M \mid M\,N \mid \mathsf{let!}\; x = M \;\mathsf{in}\; N \mid \mathsf{raise}\, M \mid e,$$

$\mathsf{let!}\; x = M \;\mathsf{in}\; N$ denotes a *strict* let expression, with the obvious intended semantics. It is included mainly to allow us to easily express the strictness transformation in section 5. Exceptions may be raised with $\mathsf{raise}\, M$, and e ranges over values of the `Exception` datatype:

$$\texttt{data Exception} \;=\; \texttt{TypeError} \mid \texttt{UserError}\; \mathit{String} \cdots$$

The language is untyped. The `TypeError` exception is used to signal type errors arising from applying exceptions or raising functions. `UserError` is self-explanatory and may be used to help define the standard Haskell function `error`:

$$\texttt{error msg} \;=\; \mathsf{raise}\,(\texttt{UserError msg})$$

Other exceptions, like `DivideByZero`, or `OverFlow` may easily be added as the need arises. Throughout, U and V will range over all values (*i.e.* lambda expressions or elements of `Exception`).

We define two forms of convergence: *normal* convergence, where evaluation terminates without an exception being raised, and *exceptional convergence*, where evaluation is brought to an abrupt halt due to the raising of an exception.

3.1 Normal Convergence

We define convergence in the functional core via a standard inductively-defined convergence relation:

$$V \Downarrow V \;\; (\mathit{Value}_{\Downarrow}) \qquad \frac{M \Downarrow \lambda x.M' \quad M'[{}^{N}\!/\!{}_{x}] \Downarrow V}{M\,N \Downarrow V}\;(\mathit{App}_{\Downarrow})$$

$$\frac{M \Downarrow U \quad N[{}^{U}\!/\!{}_{x}] \Downarrow V}{\mathsf{let!}\ x = M\ \mathsf{in}\ N \Downarrow V}\quad(\mathit{Strict\ Let}_{\Downarrow})$$

where $M \Downarrow V$ should be read as "closed term M converges to closed value V". Lambda expressions and elements of the `Exception` datatype converge immediately. Applications are evaluated in normal order, and strict lets are evaluated eagerly. Note that raise has no rule. This is as it should be: the raising of an exception is certainly not normal convergence!

3.2 Exceptional Convergence

We will write $M \uparrow e$ to mean that closed term M converges exceptionally with, or raises, exception e. Unusually, $\uparrow$ is defined *co-inductively.* The rules are labelled with "—" to indicate this fact.

We are now able to give a semantics to raise:

$$\frac{M \Downarrow e}{\mathsf{raise}\,M \uparrow e} - (\mathit{Raise}_{\uparrow_1}) \qquad \frac{M \uparrow e}{\mathsf{raise}\,M \uparrow e} - (\mathit{Raise}_{\uparrow_2})$$

raise can only converge exceptionally, but may do so in more than one way. For example, raise M will either raise the exception to which M evaluates, or raise any exception that arises during the evaluation of M. This latter behaviour is common to any strict function or constructor.

Type errors are signalled by the `TypeError` exception:

$$\frac{M \Downarrow \lambda x.N}{\mathsf{raise}\,M \uparrow \mathtt{TypeError}} - (\mathit{Raise}_{\uparrow_0}) \qquad \frac{M \Downarrow e}{M\,N \uparrow \mathtt{TypeError}} - (\mathit{App}_{\uparrow_0})$$

An application may raise an exception if its function does, or an exception is raised after substitution:

$$\frac{M \uparrow e}{M\,N \uparrow e} - (\mathit{App}_{\uparrow_1}) \qquad \frac{M \Downarrow \lambda x.M' \quad M'[{}^{N}\!/\!{}_{x}] \uparrow e}{M\,N \uparrow e} - (\mathit{App}_{\uparrow_2})$$

This is not the whole story; we are aiming for flexibility. One of the most crucial transformations in any compiler for a lazy, functional language is the strictness transformation, in which $f\,N$ is transformed to

$$\mathsf{let!}\ x = N\ \mathsf{in}\ f\,x$$

when f is discovered to be strict—in this context we define strict to mean that $f\,(\mathsf{raise}\,e)$ can raise e, for every exception e. Therefore, our notion of exceptional convergence must allow different evaluation orders: in $M\,N$, N may be evaluated before M! This motivates the final rule for application:

$$\frac{M \not\Downarrow \quad N \uparrow e}{M\,N \uparrow e} \quad (App_{\uparrow_3})$$

where $M \not\Downarrow$ is short for $\neg\exists V.M \Downarrow V$. Note that we only allow exceptional convergence in the argument to lead to exceptional convergence in the application as a whole when the function also converges exceptionally. This prevents erroneous conclusions such as

$$(\lambda x.3)\,(\mathsf{raise}\,e) \uparrow e.$$

Strict lets are similar to applications. The first two rules are analogous.

$$\frac{M \uparrow e}{\mathsf{let!}\ x = M\ \mathsf{in}\ N \uparrow e} \quad (Strict\ Let_{\uparrow_1}) \qquad \frac{M \Downarrow V \quad N[{}^{V}\!/_{x}] \uparrow e}{\mathsf{let!}\ x = M\ \mathsf{in}\ N \uparrow e} \quad (Strict\ Let_{\uparrow_2})$$

We need to be a bit clever for the analogue of the third rule. Consider the strict let expression $\mathsf{let!}\ x = M\ \mathsf{in}\ N$. If M has exceptional behaviour, then we must allow any exceptional behaviour in the body of let, by analogy with $(App_{\uparrow_3})$. The problem is that x may occur free in N, but since M hasn't converged we have nothing to which to bind x. We need some way of discovering the exceptional behaviours of N *that are independent of* x.

Our solution follows an idea from [PJRH+99]. We bind x to an auxiliary term $\mathbb{0}$ with no behaviour: it neither evaluates to any value or raises any exceptions[1]. Now, if $N[{}^{\mathbb{0}}\!/_{x}]$ raises an exception, it does so independent of the behaviour of x.

$$\frac{M \not\Downarrow \quad N[{}^{\mathbb{0}}\!/_{x}] \uparrow e}{\mathsf{let!}\ x = M\ \mathsf{in}\ N \uparrow e} \quad (Strict\ Let_{\uparrow_3})$$

Formally, we now read the $M \Downarrow V$ and $M \uparrow e$ judgements as defined on closed terms M drawn from an extended term grammar which includes the $\mathbb{0}$ term.

Divergence is Exceptional Convergence.

The rules for exceptional convergence have a non-trivial inductive interpretation: the behaviour of terms which can raise exceptions after a finite amount of computation, like $\mathsf{let!}\ x = \mathsf{raise}\,e\ \mathsf{in}\ M$ or $\mathsf{let!}\ x = V\ \mathsf{in}\ \mathsf{raise}\,e$. Indeed, any set of rules like those above may be interpreted either inductively or co-inductively. So why take the co-inductive reading in this case?

[1] Had we not chosen to let attempts to raise non-exceptions result in `TypeError` being raised in rule $(Raise_{\uparrow_0})$, we could have defined $\mathbb{0}$ to be the term $\mathsf{raise}\,(\lambda x.x)$.

By choosing the co-inductive interpretation, we include all divergent terms, like $\Omega \stackrel{\text{def}}{=} (\lambda x.x\,x)\,(\lambda x.x\,x)$, and terms whose divergence depends upon exceptional behaviour, like $(\mathsf{raise}\,e)\,\Omega$. The co-inductive interpretation stipulates that such divergent terms may raise *any* exception e, as we cannot refute that a divergent term raises e. This operational semantics models the interpretation of divergence in the denotational semantics of imprecise exceptions in [PJRH$^+$99]. It represents the idea often used in programming language semantics that divergence includes all erroneous behaviours.

This motivates the following definition of when closed term M diverges, written $M \Uparrow$:

$$M \Uparrow \stackrel{\text{def}}{=} \forall e.M \uparrow e.$$

3.3 Determinism and Exclusivity

An important property of normal convergence is that it is deterministic. That is

$$M \Downarrow U \wedge M \Downarrow V \implies U \equiv V.$$

But if an expression raises an exception, the semantics is deliberately vague about which exception may be raised. That is, it is entirely possible that $M \uparrow e_1$ and $M \uparrow e_2$ but $e_1 \not\equiv e_2$. It is this imprecision that validates many useful program transformations.

Moreover, when restricted to terms not containing $\mathbb{0}$, normal convergence is mutually exclusive with exceptional convergence, as stated by the following theorem.

Theorem 3.1 *For any given closed term M not containing $\mathbb{0}$, exactly one of the following statements is true:*

$$(i)\ \exists V.M \Downarrow V, \qquad (ii)\ \exists e.M \uparrow e.$$

We can prove

$$M \Downarrow V \implies \neg\exists e.M \uparrow e, \tag{3.1}$$

for arbitrary terms M and values V, by rule induction on $M \Downarrow V$. It then remains to prove that

$$M \not\Downarrow \implies \exists e.M \uparrow e. \tag{3.2}$$

for all closed $\mathbb{0}$-free M. This is harder because the existential quantification prevents us from arguing by rule co-induction on $M \uparrow e$. We encounter this kind of difficulty with co-inductive reasoning about the $\uparrow$ relation elsewhere in the sequel. We overcome this difficulty, by introducing an auxiliary, inductively defined exception relation, $\nearrow$, between terms M and finite non-empty

$$\mathbb{0} \nearrow \{\} \quad (\mathit{Stuck}_{\nearrow}) \qquad \frac{M \Downarrow e}{M\,N \nearrow \{\texttt{TypeError}\}} \quad (\mathit{App}_{\nearrow_0})$$

$$\frac{M \nearrow \mathcal{S} \quad N \Downarrow V}{M\,N \nearrow \mathcal{S}} \quad (\mathit{App}_{\nearrow_1}) \qquad \frac{M \Downarrow \lambda x.M_0 \quad M_0[N/x] \nearrow \mathcal{S}}{M\,N \nearrow \mathcal{S}} \quad (\mathit{App}_{\nearrow_2})$$

$$\frac{M \nearrow \mathcal{S}_1 \quad N \nearrow \mathcal{S}_2}{M\,N \nearrow \mathcal{S}_1 \cup \mathcal{S}_2} \quad (\mathit{App}_{\nearrow_3}) \qquad \frac{M \Downarrow \lambda x.N}{\mathsf{raise}\,M \nearrow \{\texttt{TypeError}\}} \quad (\mathit{Raise}_{\nearrow_0})$$

$$\frac{M \Downarrow e}{\mathsf{raise}\,M \nearrow \{e\}} \quad (\mathit{Raise}_{\nearrow_1}) \qquad \frac{M \nearrow \mathcal{S}}{\mathsf{raise}\,M \nearrow \mathcal{S}} \quad (\mathit{Raise}_{\nearrow_2})$$

$$\frac{M \nearrow \mathcal{S} \quad N[\mathbb{0}/x] \Downarrow V}{\mathsf{let!}\ x = M\ \mathsf{in}\ N \nearrow \mathcal{S}} \quad (\mathit{Strict\ Let}_{\nearrow_1}) \qquad \frac{M \Downarrow V \quad N[V/x] \nearrow \mathcal{S}}{\mathsf{let!}\ x = M\ \mathsf{in}\ N \nearrow \mathcal{S}} \quad (\mathit{Strict\ Let}_{\nearrow_2})$$

$$\frac{M \nearrow \mathcal{S}_1 \quad N[\mathbb{0}/x] \nearrow \mathcal{S}_2}{\mathsf{let!}\ x = M\ \mathsf{in}\ N \nearrow \mathcal{S}_1 \cup \mathcal{S}_2} \quad (\mathit{Strict\ Let}_{\nearrow_3})$$

Fig. 1. The rules defining $\nearrow$.

sets $\mathcal{S}$ of raised exceptions. The meaning of the judgement $M \nearrow \mathcal{S}$ is that M raises the exceptions in $\mathcal{S}$ and M doesn't diverge. It is defined by the following set of rules. This relation also plays a fundamental rôle in the development of the operational theory of applicative simulation in section 4.

Lemma 3.2 *For all closed terms M,*

$$M \uparrow e \iff (\exists \mathcal{S} \ni e. M \nearrow \mathcal{S}) \vee M \Uparrow.$$

Proof. The forward implication is equivalent to:

$$M \uparrow e \wedge \neg(\exists \mathcal{S} \ni e. M \nearrow \mathcal{S}) \implies \forall e'. M \uparrow e'.$$

This we prove by rule co-induction on $M \uparrow e'$.

For the reverse implication it suffices to show that $M \nearrow \mathcal{S} \implies \forall e \in \mathcal{S}. M \uparrow e$, which we prove by rule induction on $M \nearrow \mathcal{S}$. □

Now we see that (3.2) is equivalent to $M \not\Downarrow \wedge \neg(\exists \mathcal{S}. M \nearrow \mathcal{S}) \implies \forall e. M \uparrow e$, for all closed $\mathbb{0}$-free terms M, and this can be proved by rule co-induction on $M \uparrow e$. The proof uses the easily established fact that, whenever $M \Downarrow V$, if M doesn't contain $\mathbb{0}$, the same is true of V.

4 Refinement and Equivalence

The preceding section specified the meaning of program terms. However, compilers work by transforming one term into a semantically equivalent, but perhaps more efficient one, so we need to know precisely what *equivalence* means, and we need usable techniques to prove that two terms are equivalent.

In practice, precise equivalence is over-restrictive. Suppose a term can raise either of the exceptions e_1 or e_2. Arguably, it would be fine for a compiler to replace it with an expression that can raise only e_1, especially if the latter was more efficient. The new program doesn't have exactly the same meaning as the old one, but we argue that the change is legitimate. Why? Because the only way that we provide to catch an exception is to make a non-deterministic choice from the set, so the new program will exhibit behaviour that is always possible from the old program. (The new program will never deliver e_2, but the old one *need* never deliver e_2, depending on how the non-deterministic choice goes.)

In short, to give maximum freedom to the compiler (a good thing, since it may enable it to generate better code) we want to let it *refine* a program; that is, to transform it to a new program that refines, but is not necessarily equivalent to, the original program. A term M is refined by N, written $M \sqsubseteq\!\!\!\sim N$, if they have identical convergent behaviours, and any exceptional behaviour exhibited by N can be mimicked by M. In this section we make this definition precise, and we explain how to prove such a relationship.

To this end, we first formalise this idea of refinement as a suitable Morris-style contextual refinement preorder. This constitutes the prior notion of refinement: it's what we *really* mean by refinement. The main result of this section allows us to establish contextual refinement by showing a much simpler relationship, called *refinement similarity.* This is justified because refinement similarity is a precongruence. We sketch the proof of the precongruence of refinement similarity via the nigh-standard method due to Howe [How96].

We close with examples of the use of refinement simulation. We establish the validity of beta-laws and the strictness transformation for our language.

4.1 Contextual Refinement and Equivalence

Program contexts are usually introduced as "programs with holes", the intention being that a closed expression is to be "plugged into" all of the holes in the context. We will use contexts of the form

$$\mathbb{C}, \mathbb{D} \quad ::= \quad x \mid \lambda x.\mathbb{C} \mid \mathbb{C}\;\mathbb{D} \mid \textsf{let!}\; x = \mathbb{C} \;\textsf{in}\; \mathbb{D} \mid \textsf{raise}\,\mathbb{C} \mid e \mid \mathbb{0}.$$

Conventionally, the prior notion of observational equivalence is defined contextually; we say M is equivalent to N whenever, for all program contexts $\mathbb{C}$ such that both $\mathbb{C}[M]$ and $\mathbb{C}[N]$ are closed,

$$\mathbb{C}[M]\Downarrow \iff \mathbb{C}[N]\Downarrow$$

where the notation $M\Downarrow$ means that there exists some value V such that $M \Downarrow V$.

In the presence of imprecise exceptions, this is only half the story. The above definition would identify *all* terms that raise any kind of exception, *e.g.* Ω would be identified with raise (`UserError "No such element"`). In other words, a theory based upon the conventional definition would be oblivious to exceptions; we might as well not have added imprecise exceptions at all!

We define *contextual refinement* in such a way that it includes the above definition, but is also sensitive to exceptional behaviour.

Definition 4.1 M is *contextually refined by* N, written $M \sqsubseteq\!\!\!\!\sim N$, if, for all program contexts $\mathbb{C}$ such that both $\mathbb{C}[M]$ and $\mathbb{C}[N]$ are closed,

$$\begin{aligned} \mathbb{C}[M] \Downarrow &\iff \mathbb{C}[N] \Downarrow \\ \wedge\, \forall e.\mathbb{C}[N] \uparrow e &\implies \mathbb{C}[M] \uparrow e. \end{aligned}$$

Contextual equivalence, denoted $\cong$, is mutual contextual refinement.

Remember that we intend that if $M \sqsubseteq\!\!\!\!\sim N$, then it is legitimate for a compiler to replace M by N. Intuitively, a term is contextually refined by another if the latter has identical convergent behaviour to the former and the latter does not introduce exceptional behaviours not already present in the former. Another way of looking at it is that non-determinism (derived from possible exceptional behaviour) is not increased when moving upwards in a $\sqsubseteq\!\!\!\!\sim$-chain. This definition allows an implementation to decrease non-determinism by making choices. For instance, we shall see that by this definition $M \sqsubseteq\!\!\!\!\sim \mathsf{raise}\, e$ if $M \uparrow e$, regardless of whether M can also raise other exceptions.

4.2 Refinement Similarity and Bisimilarity

In the operational techniques community, one typically presents similarities and bisimilarities in terms of simulation functionals and other auxiliary relational operators. We will first present refinement similarity and bisimilarity in a direct fashion, deferring the more standard definitions (involving auxiliary relational operators) until section 4.3. We then state the main result of this section, that refinement similarity and bisimilarity coincide with contextual refinement and contextual equivalence.

Given a relation R between closed terms, its *open extension*, written R^o, is the relation between arbitrary terms M and M' such that $M\sigma\ R\ M'\sigma$ for every closing substitution σ for the free variables in M and M'.

We define *refinement similarity*, written $\lesssim$, as the greatest relation satisfying the following rule:

$$\frac{\begin{array}{c} \forall M_0.M \Downarrow \lambda x.M_0 \implies \exists N_0.N \Downarrow \lambda x.N_0 \wedge M_0 \lesssim^o N_0 \\ \forall N_0.N \Downarrow \lambda x.N_0 \implies \exists M_0.M \Downarrow \lambda x.M_0 \wedge N_0 \lesssim^o M_0 \\ \forall e.M \Downarrow e \iff N \Downarrow e \\ \forall e.N \uparrow e \implies M \uparrow e \end{array}}{M \lesssim N} \;-\qquad (\mathit{Ref\ Sim\ Def})$$

(Again, the $-$ indicates that this is a co-inductive definition.) It says that if all convergent behaviours of M are related by $\lesssim$ to convergent behaviours of

N (and *vice versa*), and if all exceptional behaviours of N can be matched by exceptional behaviours of M, then $M \lesssim N$. It is a simple matter to convince oneself that $\lesssim$ and $\lesssim^o$ are reflexive and transitive. Also, terms whose behaviour depends upon $\mathbb{0}$ are identified by $\lesssim$, since all of the premises of (*Ref Sim Def*) hold trivially for them.

Bisimilarity, written $\sim^o$, is mutual refinement similarity.

A given relation R is *compatible* if whenever $M\ R\ N$, we have that $\mathbb{C}[M]\ R\ \mathbb{C}[N]$ for all contexts $\mathbb{C}$. (We will give an equivalent definition of compatibility in section 4.3.) Any compatible equivalence is a congruence; any compatible preorder is a precongruence.

An important property of $\lesssim^o$ is that it is compatible.

Lemma 4.2 *$\lesssim^o$ is compatible.*

This in turn leads to the main result of this section: we can use $\lesssim^o$ to establish contextual refinement and $\sim^o$ to establish contextual equivalence.

Theorem 4.3 *(i) $M \sqsubseteq\!\!\!\sim N \iff M \lesssim^o N$.*
(ii) $M \cong N \iff M \sim^o N$.

Proof. It suffices to prove (i); (ii) is then immediate.

Suppose $M \lesssim^o N$. We must show $M \sqsubseteq\!\!\!\sim N$. Since $\lesssim^o$ is compatible, we know that for any context $\mathbb{C}$ for which $\mathbb{C}[M]$ and $\mathbb{C}[N]$ are closed, $\mathbb{C}[M] \lesssim \mathbb{C}[N]$. Therefore, if $\mathbb{C}[N] \Downarrow$ then $\mathbb{C}[M] \Downarrow$, and vice versa, and if $\mathbb{C}[N] \uparrow e$ then $\mathbb{C}[M] \uparrow e$. This is exactly the definition of $\sqsubseteq\!\!\!\sim$. Hence $\lesssim^o \subseteq \sqsubseteq\!\!\!\sim$.

The reverse inclusion, $\sqsubseteq\!\!\!\sim \subseteq \lesssim^o$, is proved co-inductively by showing that $\sqsubseteq\!\!\!\sim$ is an applicative error simulation and that the reciprocal relation, $\sqsubseteq\!\!\!\sim^{op}$, is an applicative convergence simulation. The proofs make use of the beta-laws in section 5 which, by the above inclusion $\lesssim^o \subseteq \sqsubseteq\!\!\!\sim$, can be established for $\sqsubseteq\!\!\!\sim$ by via refinement similarity. □

4.3 Compatibility of Refinement Similarity

In order to enable us to prove its compatibility, refinement similarity will be decomposed in terms of two applicative similarities. The first is essentially the same as the applicative similarity of [Abr90,How96]. The second is obtained from the first by adding an extra clause for exceptional behaviour. First, we define a useful auxiliary notion.

4.3.1 Compatible Refinement

We use the notion of the *compatible refinement* of a given relation to define what it means for a relation on open terms to be a congruence. It is also used to define the simulation functional upon which our notion of similarity will be based. If R is a binary relation over terms, then its compatible refinement, $\widehat{R}$, is defined by the rules:

$$x \mathrel{\widehat{R}} x \qquad e \mathrel{\widehat{R}} e \qquad \mathbb{0} \mathrel{\widehat{R}} \mathbb{0} \qquad \frac{M \mathrel{R} N}{\lambda x.M \mathrel{\widehat{R}} \lambda x.N} \qquad \frac{M \mathrel{R} M' \quad N \mathrel{R} N'}{M\,N \mathrel{\widehat{R}} M'\,N'}$$

$$\frac{M \mathrel{R} M' \quad N \mathrel{R} N'}{\mathsf{let!}\ x = M\ \mathsf{in}\ N \mathrel{\widehat{R}} \mathsf{let!}\ x = M'\ \mathsf{in}\ N'} \qquad \frac{M \mathrel{R} N}{\mathsf{raise}\, M \mathrel{\widehat{R}} \mathsf{raise}\, N}$$

Compatible refinement will be used in the definition of simulation below as a way of testing values, but it also provides a simple characterisation of compatibility. We can easily show that R is compatible when $\widehat{R} \subseteq R$.

4.3.2 Applicative Convergence Similarity

The applicative convergence simulation functional, is defined as follows.

Definition 4.4 Given relation R, define $[R]_{\Downarrow}$ thus

$$M \; [R]_{\Downarrow} \; N \overset{\text{def}}{=} \; \forall U.M \Downarrow U \implies \exists V.N \Downarrow V \wedge U \mathrel{\widehat{R}} V.$$

An *applicative convergence simulation* is a relation that is dense with respect to $[\cdot]^{o}_{\Downarrow}$, *i.e.* $R \subseteq [R]^{o}_{\Downarrow}$. Define applicative convergence similarity, $\lesssim^{o}_{\Downarrow}$, as the largest applicative convergence simulation.

Lemma 4.5 $\lesssim^{o}_{\Downarrow}$ *is compatible.*

We elide the proof of the compatibility of $\lesssim^{o}_{\Downarrow}$. It is a simple instance of that found in [How96].

4.3.3 Applicative Error Similarity

The applicative error simulation functional, is defined as follows.

Definition 4.6 Given relation R, define $[R]_{\uparrow}$ thus

$$\begin{aligned} M \; [R]_{\uparrow} \; N \overset{\text{def}}{=} \; & \forall U.M \Downarrow U \implies \exists V.N \Downarrow V \wedge U \mathrel{\widehat{R}} V \\ & \wedge \forall e.N \uparrow e \implies M \uparrow e. \end{aligned}$$

An *applicative error simulation* is a relation that is dense with respect to $[\cdot]^{o}_{\uparrow}$. Applicative error similarity, $\lesssim^{o}_{\uparrow}$, is the largest applicative error simulation.

Lemma 4.7 $\lesssim^{o}_{\uparrow}$ *is compatible.*

In order to prove this, we will define a candidate precongruence which by definition will be compatible *and* contain applicative error similarity. Then we will show that the candidate is an applicative error simulation. This will imply the desired result: that applicative error similarity is compatible.

The candidate congruence, written $\lesssim^{\bullet}_{\uparrow}$, is the least relation satisfying

$$\frac{M \mathrel{\widehat{\lesssim^{\bullet}_{\uparrow}}} M' \quad M' \lesssim^{o}_{\uparrow} N}{M \lesssim^{\bullet}_{\uparrow} N} \tag{4.1}$$

This relation can easily be shown to have the following properties.

Lemma 4.8 *(i)* $\lesssim^{\bullet}_{\uparrow}$ *is compatible.*

(ii) $\lesssim^{o}_{\uparrow} \subseteq \lesssim^{\bullet}_{\uparrow}$.

(iii) $\lesssim^{\bullet}_{\uparrow}; \lesssim^{o}_{\uparrow} \subseteq \lesssim^{\bullet}_{\uparrow}$.

(iv) $\lesssim^{\bullet}_{\uparrow}$ *is substitutive:* $M \lesssim^{\bullet}_{\uparrow} M' \wedge N \lesssim^{\bullet}_{\uparrow} N' \implies M[{}^{M'}\!/_{x}] \lesssim^{\bullet}_{\uparrow} N[{}^{N'}\!/_{x}]$.

This lemma states that $\lesssim^{\bullet}_{\uparrow}$ is indeed an applicative error simulation.

Lemma 4.9 $\lesssim^{\bullet}_{\uparrow} \subseteq [\lesssim^{\bullet}_{\uparrow}]^{o}_{\uparrow}$.

Proof. Since $\lesssim^{\bullet}_{\uparrow}$ is reflexive (because it is compatible) and substitutive, it suffices to prove that $M \lesssim^{\bullet}_{\uparrow} N$ implies $M\ [\lesssim^{\bullet}_{\uparrow}]_{\uparrow}\ N$ for all closed M and N. There are two parts to the proof, corresponding to the two clauses in the definition of $[\cdot]_{\uparrow}$.

(1) We are required to prove:

$$M \lesssim^{\bullet}_{\uparrow} N \wedge M \Downarrow U \implies \exists V. N \Downarrow V \wedge U \mathrel{\widehat{\lesssim^{\bullet}_{\uparrow}}} V. \tag{4.2}$$

This follows by rule induction over the judgement $M \Downarrow U$. We elide the proof (it is in fact the crucial lemma for showing the applicative convergence similarity is compatible).

(2) We are required to prove

$$M \lesssim^{\bullet}_{\uparrow} N \wedge N \uparrow e \implies M \uparrow e. \tag{4.3}$$

However, it will be sufficient to prove

$$M \lesssim^{\bullet}_{\uparrow} N \wedge M \nearrow \mathcal{S} \implies \exists \mathcal{T}. N \nearrow \mathcal{T} \wedge \mathcal{T} \subseteq \mathcal{S}. \tag{4.4}$$

since if $M \Uparrow$, then (4.3) follows trivially, and if $M \Downarrow$ then $N \Downarrow$ also, by (4.2), and in that case (4.3) holds trivially.

Suppose $M \lesssim^{\bullet}_{\uparrow} N$ and $M \nearrow \mathcal{S}$. We proceed via rule induction on the judgement $M \nearrow \mathcal{S}$. We give the application cases only. Here $M \equiv M_1\, M_2$ and, since $M \lesssim^{\bullet}_{\uparrow} N$, we know there exists an $M' \equiv M_1'\, M_2'$ such that $M_i \lesssim^{\bullet}_{\uparrow} M_i'$ for $i = 1, 2$, and $M' \lesssim^{o}_{\uparrow} N$. We may assume that M' is closed (otherwise, any closed instance of M' fits the bill, by the reflexivity and substitutivity properties of $\lesssim^{\bullet}_{\uparrow}$ and by the definition of $\lesssim^{o}$ by open extension). There are four sub-cases; we give two only.

($App_{\nearrow_0}$) $M_1 \Downarrow e$, and $M \nearrow \{\texttt{TypeError}\}$. Then by (4.2), $M_1' \Downarrow e$ also, so $M' \nearrow \{\texttt{TypeError}\}$. Since $M' \lesssim^{o}_{\uparrow} N$, we know that any exception that N can raise must be contained in this set, so we are done.

($App_{\nearrow_2}$) $M_1 \Downarrow \lambda x.M_0$ and $M_0[{}^{M_2}\!/_{x}] \nearrow \mathcal{S}$. By (4.2), $M_1' \Downarrow \lambda x.M_0'$ where $M_0 \lesssim^{\bullet}_{\uparrow} M_0'$. By substitutivity, $M_0[{}^{M_2}\!/_{x}] \lesssim^{\bullet}_{\uparrow} M_0'[{}^{M_2'}\!/_{x}]$, so by the inductive hypothesis, $M_0'[{}^{M_2'}\!/_{x}] \nearrow \mathcal{T}$ for some $\mathcal{T} \subseteq \mathcal{S}$. Therefore $M' \nearrow \mathcal{T}$, and the result follows since $M' \lesssim^{o}_{\uparrow} N$. □

Proof of Lemma 4.7. $\precsim^o_\uparrow$ is compatible.

Proof. $\precsim^o_\uparrow$ is by definition the greatest (open) applicative error simulation. Therefore, since $\precsim^\bullet_\uparrow$ is an applicative error simulation, by lemma 4.9, $\precsim^\bullet_\uparrow \subseteq \precsim^o_\uparrow$. But $\precsim^o_\uparrow \subseteq \precsim^\bullet_\uparrow$, by lemma 4.8(ii), so $\precsim^o_\uparrow = \precsim^\bullet_\uparrow$. $\precsim^\bullet_\uparrow$ is compatible, by lemma 4.8(i), so $\precsim^o_\uparrow$ is also. □

4.3.4 Compatibility of Refinement Similarity

We are now in a position to decompose refinement similarity into applicative convergence similarity and applicative error similarity.

Lemma 4.10 $M \precsim^o N \iff N \precsim^o_\Downarrow M \wedge M \precsim^o_\uparrow N.$

A similar lemma holds for bisimilarity.

Proof of Lemma 4.2. $\precsim^o$ is compatible.

Proof. Since its component parts are compatible, $\precsim^o$ is compatible. □

5 The payoff: examples of equivalences

So much for the underlying theory. It is time to return to the claim we made in the introduction, namely that we have added exceptions to a lazy language *without losing any useful program transformations.* Have we met that goal? We don't know how to prove it in general. What we do instead in this section is to take several practically-useful equivalences in the ordinary call-by-name calculus, and prove that they remain valid in our extended language.

5.1 Simple laws

The following inferences are valid for all closed terms M by (3.1) and the soundness of $\precsim$ and $\sim$:

$$\frac{M \Downarrow V}{M \cong V}\ (\Downarrow) \qquad \frac{M \uparrow e}{M \sqsubsetsim \mathsf{raise}\, e}\ (\uparrow) \qquad \frac{M \not\Downarrow}{M \sqsubsetsim \mathbb{0}}\ (\mathbb{0})$$

The first also depends upon the deterministic nature of $\Downarrow$. Two simple beta-laws are easily shown to be valid:

$$(\lambda x.M)\, N \cong M[^N/_x] \qquad (\beta)$$
$$\mathsf{let!}\ x = V\ \mathsf{in}\ M \cong M[^V/_x] \qquad (\mathsf{let!}\text{-}\beta)$$

For example, whenever $(\lambda x.M)\, N \Downarrow V$ so does $M[^N/_x]$ by $(App_\Downarrow)$, and *vice versa.* Furthermore, whenever $(\lambda x.M)\, N \uparrow e$, so does $M[^N/_x]$, by $(App_{\uparrow_2})$, and *vice versa.* Therefore, $(\lambda x.M)\, N \sim M[^N/_x]$, and we have that $(\lambda x.M)\, N \sim M[^N/_x]$, when $(\lambda x.M)\, N$ and $M[^N/_x]$ are closed. This extends to open terms since $\sim$ is substitutive, and the result follows.

As a result of (β) we get that $\cong$ is substitutive:

$$\begin{array}{rll} M \cong M' \wedge N \cong N' & \Longrightarrow (\lambda x.M)N \cong (\lambda x.M')N' & \text{congruence} \\ & \Longrightarrow M[{}^{N}\!/_{x}] \cong M'[{}^{N'}\!/_{x}] & (\beta) \end{array}$$

This is also true of $\sqsubseteq$ with $\sim$ beneath, by similar reasoning.

5.2 Commuting independent evaluations

The next example is more substantial. It captures the essence of the commutativity of the + example from the introduction.

Proposition 5.1 *Provided x and y are distinct and do not occur free in either M or N,*

$$\mathsf{let!}\ x = M\ \mathsf{in}\ \mathsf{let!}\ y = N\ \mathsf{in}\ P \cong \mathsf{let!}\ y = N\ \mathsf{in}\ \mathsf{let!}\ x = M\ \mathsf{in}\ P.$$

Proof. Let *LHS* and *RHS* refer to the two expressions we are equating. As was the case for the beta-laws, it is enough to prove the equation when *LHS* and *RHS* are closed. Then M and N are also closed. We proceed via a case analysis of the convergence behaviour of M and N. There are four cases; we give two only.

- $M \Downarrow U$, and $N \Downarrow V$, for some U and V. Then

$$\begin{array}{rll} \mathit{LHS} & \cong \mathsf{let!}\ x = U\ \mathsf{in}\ \mathsf{let!}\ y = V\ \mathsf{in}\ P & (\Downarrow), \text{congruence} \\ & \cong P[{}^{U}\!/_{x}][{}^{V}\!/_{y}] & (\mathsf{let!}\text{-}\beta) \times 2 \\ & \equiv P[{}^{V}\!/_{y}][{}^{U}\!/_{x}] & U,\ V \text{ are closed and} \\ & & x,\ y \text{ are distinct} \\ & \cong \mathsf{let!}\ y = V\ \mathsf{in}\ \mathsf{let!}\ x = U\ \mathsf{in}\ P & (\mathsf{let!}\text{-}\beta) \times 2 \\ & \cong \mathit{RHS} & (\Downarrow), \text{congruence} \end{array}$$

- $M \not\Downarrow$ and $N \not\Downarrow$. Then $\mathit{LHS} \not\Downarrow$ and $\mathit{RHS} \not\Downarrow$ too. Thus it suffices to show that

$$\mathit{LHS} \uparrow e \iff \mathit{RHS} \uparrow e, \tag{5.1}$$

for all exceptions e. By the rules defining the exceptional convergence relation for strict let, we see that $\mathit{LHS} \uparrow e$ holds if and only if one of

(i) $M \uparrow e$,
(ii) $N \uparrow e$, or
(iii) $P[{}^{\mathbb{0}}\!/_{x}][{}^{\mathbb{0}}\!/_{y}] \uparrow e$

holds. But the same is true of *RHS*, so we conclude (5.1), as required. □

5.3 Transforming call by name into call by value

Our final application involves the strictness transformation. It says that transforming call-by-name (or need) into call-by-value is a refinement whenever the context involved is strict in the sense of the premise. This transformation is one of the most commonly applied in functional language compilers, and it is important that it remains valid.

Proposition 5.2 *For all M and N, then the following is a valid inference rule:*

$$\frac{\forall e. M[{}^{\mathsf{raise}\, e}\!/_{x}] \uparrow e}{M[{}^{N}\!/_{x}] \cong \mathsf{let!}\ x = N\ \mathsf{in}\ M}$$

The premise says that M is strict in x; that is, if x is replaced by $\mathsf{raise}\, e$ in M, the resulting term is sure to be able to raise e. The bottom line says that a call-by-name binding of x to an arbitrary term N (denoted by substitution) is equivalent to a call-by-value binding (denoted by $\mathsf{let!}$).

Proof. Assume $M[{}^{\mathsf{raise}\, e}\!/_{x}] \uparrow e$ for all e, and consider N. Again, it is enough to prove the result when both $M[{}^{N}\!/_{x}]$ and $\mathsf{let!}\ x = N\ \mathsf{in}\ M$ are closed, since $\precsim$ and $\cong$ are substitutive. Then N must be closed. We split the argument into two cases:

- $N \Downarrow V$, for some V. Then

$$\begin{aligned} \mathsf{let!}\ x = N\ \mathsf{in}\ M &\cong \mathsf{let!}\ x = V\ \mathsf{in}\ M && (\Downarrow), \text{congruence} \\ &\cong M[{}^{V}\!/_{x}] && (\mathsf{let!}\text{-}\beta) \\ &\cong M[{}^{N}\!/_{x}] && \text{substitutivity,} \end{aligned}$$

  as required.

- $N \not\Downarrow$. We prove that $M[{}^{N}\!/_{x}]$ and $\mathsf{let!}\ x = N\ \mathsf{in}\ M$ are mutual refinements.

  First, we show that $M[{}^{N}\!/_{x}] \precsim \mathsf{let!}\ x = N\ \mathsf{in}\ M$. Suppose $\mathsf{let!}\ x = N\ \mathsf{in}\ M \uparrow e$. This is derived either by (*Strict Let*$_{\uparrow_1}$) because $N \uparrow e$, or by (*Strict Let*$_{\uparrow_3}$) because $M[{}^{\mathbb{0}}\!/_{x}] \uparrow e$. In the first case, $M[{}^{N}\!/_{x}] \uparrow e$ is immediate from the strictness assumption. In the second case, we use that by ($\mathbb{0}$) we have that $N \precsim \mathbb{0}$. Therefore, $M[{}^{N}\!/_{x}] \precsim M[{}^{\mathbb{0}}\!/_{x}]$. Hence $M[{}^{\mathbb{0}}\!/_{x}] \uparrow e$ implies that $M[{}^{N}\!/_{x}] \uparrow e$, and we are done.

  Next, we show that $\mathsf{let!}\ x = N\ \mathsf{in}\ M \precsim M[{}^{N}\!/_{x}]$. This direction is harder. We need two facts:

$$\begin{aligned} M[{}^{N}\!/_{x}] \Downarrow V &\implies \exists U.\ V = U[{}^{N}\!/_{x}] \wedge \\ &\qquad \forall N'. M[{}^{N'}\!/_{x}] \Downarrow U[{}^{N'}\!/_{x}], && (5.2) \\ M[{}^{N}\!/_{x}] \uparrow e \wedge \neg(N \uparrow e) &\implies M[{}^{\mathbb{0}}\!/_{x}] \uparrow e. && (5.3) \end{aligned}$$

  We establish these facts for arbitrary terms M; (5.2) by rule induction on

$M[{}^N\!/_x] \Downarrow V$, and (5.3) by rule co-induction on $M[{}^{\mathbb{0}}\!/_x] \uparrow e$. From (5.2), the strictness assumption and (3.1) we conclude that $M[{}^N\!/_x] \not\Downarrow$, so it suffices to show that $M[{}^N\!/_x] \uparrow e$ implies $\mathsf{let!}\ x = N\ \mathsf{in}\ M \uparrow e$ for all e. But this follows easily from (5.3), (*Strict Let*$_{\uparrow_1}$) and (*Strict Let*$_{\uparrow_3}$), and we are done. □

6 Extensions

The language described thus far is not particularly expressive (it lacks data constructors, case expressions, integers, and primitives among other things). It is also not so useful: we can raise exceptions with ease, but cannot catch them! In this section, we sketch how the language and its semantics may be extended to allow for more realistic language constructs, I/O operations, catching and handling of exceptions, and primitives for concurrency. The details of these extensions may be found in a forthcoming longer version of this paper.

6.1 Data Constructors, etc.

We can easily add the following syntactic constructs to the language:

$$\cdots \mid c\,M_1 \cdots M_n \mid \mathsf{case}\ M\ \mathsf{of}\ \{\, c_i\,\vec{x}_i \to N_i \,\} \mid \ulcorner i \urcorner \mid M\ op\ N \mid \mathsf{fix}\,M$$

where c ranges over a set of constructor names (not renameable and disjoint from variable names), $\ulcorner i \urcorner$ is a distinguished value corresponding to integer i, and op ranges over a set of primitive operators. fix is explicit recursion, and case allows us to match on constructors.

The normal convergence semantics rules for these constructs are standard, and the exceptional convergence semantics rules are straightforward adaptions of those that were presented in section 3. For example, the three rules describing the exceptional behaviour of case expressions are very similar to those for strict let expressions; this is the analogue of (*Strict Let*$_{\uparrow_3}$):

$$\frac{M \not\Downarrow \quad N_j[{}^{\vec{\mathbb{0}}}\!/_{\vec{x}_j}] \uparrow e}{\mathsf{case}\ M\ \mathsf{of}\ \{\, c_i\,\vec{x}_i \to N_i \,\} \uparrow e} - \qquad (Case_{\uparrow_3})$$

The proofs in section 4 can also be extended to take the new constructs into account in a straightforward manner.

6.2 The I/O Monad

We can add the I/O monad to the language by extending the latter with the following constructs:

$$\cdots \mid \mathsf{return}\,M \mid M \gg\!= N \mid \mathsf{getChar} \mid \mathsf{putChar}\,M \mid \mathsf{raiseIO}\,M.$$

$\mathsf{raiseIO}\,M$ represents a computation that when performed will raise the exception indicated by M. External interrupts (such as the user typing Control-C) are represented by $\mathsf{raiseIO}$, but the programmer may also use it. Any other desired I/O operations may be added similarly. I/O computations are left almost untouched by the evaluation semantics, since they are treated as constructors. Following [PJGF96], we give a transition style semantics to the I/O operations. Two example axioms of the monadic transition semantics are

$$\begin{gathered}(\mathsf{return}\,M) \gg\!= N \to N\,M \\ (\mathsf{raiseIO}\,M) \gg\!= N \to \mathsf{raiseIO}\,M.\end{gathered}$$

All transitions take place within evaluation contexts of the following form:

$$\mathbb{E} \ ::= \ [\cdot] \mid \mathbb{E} \gg\!= M.$$

An external interrupt replaces the current computation with a $\mathsf{raiseIO}$:

$$\mathbb{E}[M] \xrightarrow{\lightning e} \mathbb{E}[\mathsf{raiseIO}\,e]$$

where $\mathbb{E}$ is the maximal evaluation context of $\mathbb{E}[M]$ (*i.e.* M is not of the form $M_1 \gg\!= M_2$). We have labelled the transition with $\lightning e$ to indicate that an aysnchronous exception has occurred. We are still unable to catch any exceptions at this point, but now that we have the I/O monad, help is at hand.

6.3 Catching and Handling Exceptions

We add two new operations to the I/O monad:

$$\cdots \mid \mathsf{getException}\,M \mid \mathsf{getExceptionIO}\,M.$$

The former catches and reifies exceptions that are raised during normal evaluation (like division by zero), while the latter catches and reifies I/O exceptions and external interrupts. Here's how $\mathsf{getException}$ works:

$$\frac{M \Downarrow V}{\mathsf{getException}\,M \to \mathsf{return}\,(\mathsf{Ok}\,V)}$$
$$\frac{M \uparrow e}{\mathsf{getException}\,M \to \mathsf{return}\,(\mathsf{Bad}\,e)}$$

where Ok and Bad are tags that may be inspected by exception handlers. $\mathsf{getExceptionIO}$ is similar. We also need to extend evaluation contexts:

$$\mathbb{E} \ ::= \ [\cdot] \mid \mathbb{E} \gg\!= M \mid \mathsf{getExceptionIO}\,\mathbb{E}.$$

6.4 Concurrent Haskell with Exceptions

It is also relatively simple to combine exceptions with Concurrent Haskell [PJGF96]. Add the following I/O operations:

$$\cdots \mid \mathsf{forkIO}\, M \mid \mathsf{newMVar}\, M \mid \mathsf{takeMVar}\, M \mid \mathsf{putMVar}\, M\, N \mid \mathsf{signalIO}\, M\, N.$$

forkIO spawns a new process, containing I/O computation M, and returns a thread identifier. $\mathsf{newMVar}$, $\mathsf{takeMVar}$, and $\mathsf{putMVar}$ concern shared, synchronised variables. The new feature relative to [PJGF96] is $\mathsf{signalIO}$, which allows one thread to raise an exception in another.

Now we can extend the transition system given above to work on processes of the following form:

$$P, Q \;::=\; \mathbb{0} \mid \langle\!\langle M \rangle\!\rangle_t \mid \langle\rangle_n \mid \langle M \rangle_n \mid \nu x.P \mid P \,||\, Q,$$

comprising a nil process, a thread of computation named t, empty and full *MVar*s, restriction, and parallel composition. For example, here is the transition for forkIO:

$$\langle\!\langle \mathbb{E}[\mathsf{forkIO}\, M] \rangle\!\rangle_t \rightarrow \nu u.\langle\!\langle \mathbb{E}[\mathsf{return}\, u] \rangle\!\rangle_t \,||\, \langle\!\langle M \rangle\!\rangle_u, \quad u \notin \mathit{fn}\,(\mathbb{E}, M).$$

7 Conclusions and Future Work

We have built an operational theory for imprecise exceptions that corresponds very closely to the denotational semantics of [PJRH$^+$99]. By showing that refinement similarity, a simple and effective means of establishing refinement and equivalence based upon applicative simulation, coincides with respect to contextual refinement, we were able to verify most of the standard call-by-name equations. The advantage of having an operationally-based theory is that we can more readily extend the language; in particular, adding concurrency is easy.

One shortcoming of the present theory is that the `Exception` type is flat: exceptions have no structure. Allowing exceptions to be arbitrarily complex doesn't appear to pose any significant problems, and is certainly something that should be pursued.

We have only discussed call by *name* in this paper, although part of our motivation for using an operational semantics is to make it possible to be precise about call by *need*, following [MS99]. However, giving an operational semantics for imprecise exceptions and call-by-need is not a trivial matter, since care must be taken to ensure that presence of sharing doesn't interfere with the non-determinism inherent in imprecise exceptions. Another difficulty is the fact that there is no known congruent applicative bisimilarity for call-by-need, so perhaps the theory would need to be developed via an abstract machine instead (along the lines of [MS99]).

Acknowledgements.

The first author gratefully acknowledges the hospitality of the Microsoft Research Lab (Cambridge), where this work was begun, and the Oregon Graduate Institute, where this work was completed.

References

[Abr90] S. Abramsky, *The lazy lambda calculus*, Research Topics in Functional Programming (D. Turner, ed.), Addison-Wesley, 1990.

[CC91] P. Cousot and R. Cousot, *Inductive definitions, semantics and abstract interpretation*, Proc. POPL'91, ACM Press, January 1991.

[Gor94] A. D. Gordon, *Functional programming and input/output*, Distinguished Dissertations in Computer Science, Cambridge University Press, 1994.

[HM95] J. Hughes and A. K. Moran, *Making choices lazily*, Proc. FPCA'95, ACM Press, June 1995, pp. 108–119.

[How96] D. Howe, *Proving congruence of bisimulation in functional programming*, Information and Computation **124** (1996), no. 2, 103–112.

[Las98] S. B. Lassen, *Relational reasoning about functions and nondeterminism*, Ph.D. thesis, Department of Computer Science, University of Aarhus, December 1998, BRICS Dissertation Series DS-98-2.

[Mor98] A. K. Moran, *Call-by-name, call-by-need, and McCarthy's Amb*, Ph.D. thesis, Department of Computing Science, Chalmers University of Technology, September 1998.

[MS99] A. K. Moran and D. Sands, *Improvement in a lazy context: An operational theory for call-by-need*, Proc. POPL'99, ACM, 1999.

[MSC99] A. K. Moran, D. Sands, and M. Carlsson, *Erratic Fudgets: A semantic theory for an embedded coordination language*, Coordination '99, LNCS 1594, Springer-Verlag, 1999.

[PJGF96] S. L. Peyton Jones, A. D. Gordon, and S. Finne, *Concurrent Haskell*, Proc. POPL'96, ACM Press, 1996, pp. 295–308.

[PJRH$^+$99] S. L. Peyton Jones, A. Reid, C. A. R. Hoare, S. Marlow, and F. Henderson, *A semantics for imprecise exceptions*, ACM SIGPLAN Notices **34** (1999), no. 5, 25–36, Proc. of PLDI'99.

[PJW93] S. L. Peyton Jones and P. Wadler, *Imperative functional programming*, Proc. POPL'93, ACM Press, 1993, pp. 71–84.

[Plo75] G. D. Plotkin, *Call-by-name, call-by-value and the λ-calculus*, Theoretical Computer Science **1** (1975), 125–159.