Exceptionally Safe Futures

Armand Navabi and Suresh Jagannathan

Purdue University, Department of Computer Science
{anavabi,suresh}@cs.purdue.edu

Abstract. A *future* is a well-known programming construct used to introduce concurrency to sequential programs. Computations annotated as futures are executed asynchronously and run concurrently with their continuations. Typically, futures are *not* transparent annotations: a program with futures need not produce the same result as the sequential program from which it was derived. *Safe* futures guarantee a future-annotated program produce the same result as its sequential counterpart. Ensuring safety is especially challenging in the presence of constructs such as exceptions that permit the expression of non-local control-flow. For example, a future may raise an exception whose handler is in its continuation. To ensure safety, we must guarantee the continuation does not discard this handler regardless of the continuation's own internal control-flow (e.g. exceptions it raises or futures it spawns). In this paper, we present a formulation of safe futures for a higher-order functional language with first-class exceptions. Safety can be guaranteed dynamically by stalling the execution of a continuation that has an exception handler *potentially* required by its future until the future completes. To enable greater concurrency, we develop a static analysis and instrumentation and formalize the runtime behavior for instrumented programs that allows execution to discard handlers *precisely* when it is safe to do so.

1 Introduction

A *future* [3] provides a simple way for programmers to introduce concurrency to sequential programs. When executed, a computation annotated as a future yields a *placeholder* and introduces an asynchronous thread of control whose result is stored within the associated placeholder. When the computation following the future (its *continuation*) requires the future's value, it performs a *touch* or *claim* operation on the placeholder. A claim action acts as a synchronization barrier, forcing the continuation to block until the future yields a result. For programs with no side-effects, a future-annotated program exhibits the same observable behavior as the original sequential version. To preserve deterministic behavior equivalent to that of the original sequential program in the presence of side-effects requires additional machinery.

Consider the code example in Figure 1. Function `f` takes an integer argument `x`. If `x` is even, it returns the result of applying `g` to the value stored in reference `r`. If `x` is odd, it stores the result of `g (x)` in `r` and returns `x`. Variable `a` is bound to the result of a future-annotated computation (line 5). Thus computation `f (m)` is executed concurrently with its continuation. The continuation spawns future `f (n)` to be bound to `b` (line 6), which is evaluated concurrently with call `f (p)` (line 7). Thus, the three calls to function `f` will be executed concurrently. *Safe* futures require that concurrent execution of these calls must adhere to the dependences imposed by sequential evaluation: a read of reference `r` performed in one call must not witness a write to `r` by a later one, and a write to `r` by one call must be witnessed by a read of `r` in a later call.

```
let val g = fn x => (* side-effect free computation *)          1
    val r = ref 0                                               2
    val f = fn x => if ((x mod 2) = 0) then g (!r)              3
                    else (r := g (x); x)                        4
in let val a = future (f (m))                                   5
       val b = future (f (n))                                   6
   in f (p)                                                     7
   end                                                          8
end                                                             9
```

Fig. 1. Safe futures in the presence of mutable references

Safety can be guaranteed using both dynamic [10] and static [9] techniques. For example, compiler inserted barriers supported by a lightweight runtime can be used to enforce dependences defined by the sequential semantics [9]. Figure 2 illustrates how function `f` can be rewritten based on an interprocedural control-flow analysis. The read on line 3 is preceded by barrier `ALLOWED(L3)`, which completes only once all futures in the logical past have granted permission by performing a `GRANT(L3)` operation. A future can grant permission for condition `L3` once it has entered a branch in which no further conflicting write access to `r` will be performed, or it has completed its final write to `r`. Thus in the true branch of function `f`, the future will immediately grant on `L3` allowing concurrent execution of its continuation to read `r` because the future will not write it. In the false branch, the future will only grant after it has written to `r`, ensuring its continuation will witness its write. Note that similar instrumentation is required to force the write on line 7 to wait for its futures (i.e. computations which execute in the logical past) to read `r`, but has been omitted in Figure 2 for brevity.

```
val f =                                                         1
     fn x => if ((x mod 2) = 0)                                 2
             then let val tmp = (GRANT(L3); ALLOWED(L3); !r)    3
                  in g (tmp)                                    4
                  end                                           5
             else let val tmp = g (x)                           6
                      val _ = (r := tmp; GRANT(L3))             7
                  in x                                          8
                  end                                           9
```

Fig. 2. Barrier Instrumentation

Given the instrumentation presented in Figure 2, consider the resulting runtime schedule for an execution where $m = 13$, $n = 4$ and $p = 2$. Since m is odd, the call `f (m)` will write to `r`. Both `f (n)` and `f (p)` will not write to `r`, and therefore both immediately grant on condition `L3` notifying their continuations that they will not change the value of reference `r`. Before reading the value of `r`, they perform an `ALLOWED` operation on condition `L3`. The first future computation, `f (m)` is logically ordered before both computations, and therefore their `ALLOWED` barriers must wait for the future's `GRANT`. The future computation grants *after* it has written to `r` ensuring the currently executing calls to `f` read the value of `r` that is consistent with a sequential execution.

Unfortunately, the presence of mutable references is not the only means by which sequential behavior can be compromised. Exceptions and related abstractions that introduce non-local control-flow introduce challenging complications. In the presence of exceptional control-flow, a future may raise an exception whose handler is defined in its continuation. Since the future and continuation are evaluated concurrently, the continuation must not be allowed to discard a handler that may be required by the future.

Consider the code example presented in Figure 3. The example does not have mutable references. Function `f` either returns the result of applying `g` to argument `x` if `x` is odd, or raises an exception if `x` is even. Under a sequential evaluation (i.e. one with futures erased), `f (m)` and `f (n)` are evaluated to completion in that order. If future `f (m)` raises an exception that it does not internally handle (i.e. an *escaping* exception), the continuation `f (n)` is not evaluated. For example if $m = 0$ then computation `f (m)` on line 5 raises an escaping exception and computation `f (n)` is not evaluated. Instead the exception raised by the future is handled by the handler on line 6 and the program evaluates to 0.

```
let val g = fn x => (* side-effect free computation *)          1
    val f = fn x => if ((x mod 2) = 1) then g (x)               2
                    else if (x = 0) then raise ZeroException    3
                         else raise NonZeroEvenException        4
in (let b = future (f (m)) in f (n) end)                        5
    handle ZeroException => 0                                   6
         | NonZeroEvenException => 1                            7
end                                                             8
```

Fig. 3. Safe futures in the presence of exception handling

To enforce determinism in the presence of concurrent execution, constraints must be imposed on what a future's continuation may do. In a concurrent execution, future `f (m)` and continuation `f (n)` are evaluated concurrently, but if the future raises an escaping exception the continuation should have never been evaluated. This imposes constraints on continuation `f (n)`. For example, if $n = 2$, then it will raise exception `NonZeroEvenException`. If it raises the exception before its future completes, the evaluation cannot discard the `ZeroException` handler and handle `NonZeroEven-Exception` since the handlers may be required by `f (m)`. Furthermore, even if the continuation does not raise an exception and instead evaluates to a value (i.e. if $n = 3$), the continuation is not free to evaluate past the exception handlers because its future still may raise an escaping exception that requires one of the handlers.

Unfortunately, non-local control-flow introduced by exceptions makes static injection of barriers (e.g. `GRANT`, `ALLOWED`) ineffective for exploiting parallelism. To see why, consider the following example:

```
(let y = future (f (n)) in c ()) handle E => ...
```

To disallow evaluation of the continuation `c ()` from discarding the handler that may be required by future `f (n)`, `ALLOWED` barriers need to be inserted at all exit points in function `c`. The exit points consist of program points that signify successful completion of the function and any raise statement that potentially raises an escaping exception. `GRANT`'s would need to be inserted in function `f` at program points where `f` is guaranteed to no longer raise escaping exception `E` so that future `f (n)` will notify its continuation of when it is safe for it to proceed past the handler for `E`. In the presence of non-local control flow and first-class exceptions, a static analysis cannot precisely determine which static raise statements may raise escaping exceptions. Imprecision in the static analysis leads to an overly conservative injection of `ALLOWED` barriers, forcing continuations to block when it may be safe to proceed. As a result, static injection of barriers may potentially limit parallelism significantly.

Determining the earliest point during execution for which it is safe for a continuation to cross a handler boundary (i.e. discard a handler) is the focus of this paper. The context of our investigation is a higher-order functional language with first-class exceptions. We present an operational semantics that guarantees safety by stalling a continuation from discarding an exception handler before its future (or any future *it* may have created) is still executing. To enable greater concurrency, we formalize a flow-sensitive static analysis and instrumentation technique to annotate program points with possible escaping exceptions that may be reached from that point. We then define an operational semantics on instrumented programs that allows a continuation to cross a handler boundary before futures spawned in its try block have completed, if those futures and the futures they spawn (or may spawn) are guaranteed to not require the handler, as dictated by the results of the static analysis. Our results are the first to formalize the integration of safe (deterministic) futures within a language that supports first-class exceptions.

2 The Language

Figure 4 presents the syntax of a higher-order functional language Λ that has futures and first-class exceptions. The language is based on the language presented in [1]. It is an intermediate representation of an idealized functional language with futures. The language has been extended with first-class exceptions, an `exit` primitive that terminates the computation, and constructs to raise and handle exceptions. Like [1], our language does not have a touch (or claim) primitive. Instead, the parallel semantics we present transparently *touch* placeholder variables. This makes future annotations truly transparent relieving the programmer of the burden of inserting touch operations based on the

KERNEL-LANGUAGE Λ:

$$\begin{array}{rcl}
M \in \Lambda & ::= & V_l \mid x_l \mid \texttt{exit}_l\ x \mid \texttt{raise}_l\ x \\
& \mid & \texttt{try}_l\ M\ \texttt{handle}\ X \mapsto M \\
& \mid & \texttt{let}\ x = V\ \texttt{in}\ M \\
& \mid & \texttt{let}\ x = \texttt{if}\ y\ \texttt{then}\ M\ \texttt{else}\ M\ \texttt{in}\ M \\
& \mid & \texttt{let}\ x = (y\ z)\ \texttt{in}\ M \\
& \mid & \texttt{let}\ x = \texttt{future}(M)\ \texttt{in}\ M \\
V \in \mathit{Value} & ::= & c \mid \lambda\,\texttt{x}.M \mid X \\
x \in \mathit{Vars} & := & \{x, y, z, \ldots\} \\
c \in \mathit{Const} & := & \{\texttt{unit}, \texttt{true}, \texttt{false}, 0, 1, \ldots\} \\
X \in \mathit{Exception} & ::= & \texttt{Exn}_1 \mid \texttt{Exn}_2 \mid \ldots \mid \texttt{Exn}_n
\end{array}$$

Fig. 4. Language Syntax

data flow properties of the program. Although the language does not support dynamic creation of new exception values, adding such functionality does not introduce any additional complexity to our development. We make the usual assumption that all λ- and `let`-bound variables are distinct. All other terms in the language (i.e. variables, values, exit statements, ...) are given unique labels so that the static analysis and instrumentation presented in Section 4 can uniquley identify program terms.

2.1 Sequential Evaluation Semantics

Figure 5 defines the sequential semantics for programs in Λ. The semantics is defined by function F_{seq} that maps a program M to a result R where R is either a constant, a procedure (i.e., λ-term), an exception value, or `error`. The semantics erases future annotations in a program M with a runtime term that synchronously evaluates the future computation and binds its result to a variable (resulting in program $\widetilde{M}$). The evaluation rule $N \rightarrow_{seq} N'$ reduces runtime term N to a new program term N'. Evaluating `exit` V causes evaluation to terminate with result V.

3 Safe (Parallel) Dynamic Evaluation

In the parallel semantics presented in this section, the result of an incomplete future computation is represented at runtime by a placeholder. The semantics specifies concurrently evaluating future computations and enforces a global logical order on computations. As demonstrated in the semantics presented in Section 3.1, this ordering is used when a computation attempts to exit the program and when a future computation invalidates its continuation by raising an escaping exception. Logically, a future computation N_f is ordered before the computation N_c associated with its continuation. Any future computations spawned during the evaluation of N_f are also ordered before N_c. This ordering is maintained by assigning each computation an *order identifier* consisting of a real number $\texttt{r}$ and integer $\texttt{d}$. A computation with order identifier $(\texttt{r}, \texttt{d})$ is logically ordered before a computation with $(\texttt{r}', \texttt{d}')$ if $\texttt{r} < \texttt{r}'$. The integer $\texttt{d}$ in the order

$$R \in \mathit{Results} ::= c \mid \lambda x.M \mid X \mid \texttt{error}$$

$$\begin{aligned} N \in \mathit{RTTerms} ::= {} & V \mid x \mid \texttt{exit}\ V \mid \texttt{raise}\ X \mid \texttt{try}\ N\ \texttt{handle}\ X \mapsto N \\ \mid {} & \texttt{let}\ x = \texttt{if}\ V\ \texttt{then}\ N\ \texttt{else}\ N\ \texttt{in}\ N \mid \texttt{let}\ x = (V\ V)\ \texttt{in}\ N \\ \mid {} & \texttt{let}\ x = N\ \texttt{in}\ N \end{aligned}$$

$$\widetilde{M} = M[(\texttt{let}\ x = \texttt{future}(M')\ \texttt{in}\ M'')/(\texttt{let}\ x = M'\ \texttt{in}\ M'')]$$

$$F_{seq}(M) = \begin{cases} V & \texttt{if}\ \widetilde{M} \Rightarrow^*_{seq} V \\ \texttt{error} & \texttt{otherwise} \end{cases}$$

EVALUATION RULES:

$$\varepsilon \in \mathit{EvalCntxt} ::= [\,] \mid \texttt{try}\ \varepsilon\ \texttt{handle}\ X \mapsto N \mid \texttt{let}\ x = \varepsilon\ \texttt{in}\ N$$

$$\frac{N \rightarrow_{seq} N'}{\varepsilon[N] \Rightarrow_{seq} \varepsilon[N']} \qquad \frac{}{\varepsilon[\texttt{exit}\ V] \Rightarrow_{seq} V}$$

$\texttt{try}\ V\ \texttt{handle}\ X \mapsto N$	$\rightarrow_{seq} V$	(try)
$\texttt{try raise}\ X\ \texttt{handle}\ X \mapsto N$	$\rightarrow_{seq} N$	(handle)
$\texttt{try raise}\ X'\ \texttt{handle}\ X \mapsto N$	$\rightarrow_{seq} \texttt{raise}\ X'$	$(\mathit{tryraise})$
$\texttt{let}\ x = V\ \texttt{in}\ N$	$\rightarrow_{seq} N[x/V]$	(bind)
$\texttt{let}\ x = \texttt{raise}\ X\ \texttt{in}\ N$	$\rightarrow_{seq} \texttt{raise}\ X$	$(\mathit{bindraise})$
$\texttt{let}\ x = \texttt{if}\ V\ \texttt{then}\ N_1\ \texttt{else}\ N_2\ \texttt{in}\ N$	$\rightarrow_{seq} \begin{cases} \texttt{let}\ x = N_1\ \texttt{in}\ N & V = \texttt{true} \\ \texttt{let}\ x = N_2\ \texttt{in}\ N & V = \texttt{false} \end{cases}$	(if)
$\texttt{let}\ x = (V\ V')\ \texttt{in}\ N$	$\rightarrow_{seq} \texttt{let}\ x = N'[y/V']\ \texttt{in}\ N \quad V = \lambda y.N'$	(apply)

FIG. 5. SEQUENTIAL EVALUATION

identifier is used to determine how to compute new order identifiers for newly spawned computations. The primordial main computation is given order identifier $(\mathtt{0.0}, \mathtt{0})$.

Let computation N, with order identifier $(\mathtt{r}, \mathtt{d})$, evaluatee the following runtime term: $(\mathtt{let}\ x = \mathtt{future}(M)\ \mathtt{in}\ M')$. The semantics replaces computation N with two new computations N_f and N_c to evaluate the future computation M and its continuation M', respectively. N_f is given order identifier $(\mathtt{r}, \mathtt{d}+\mathtt{1})$ and N_c is given $(\mathtt{r}+\mathtt{0.5}^{\mathtt{d}}, \mathtt{d}+\mathtt{1})$. This ordering implies that N_f is logically ordered before N_c because $(\mathtt{r} < (\mathtt{r}+\mathtt{0.5}^{\mathtt{d}}))$.

Computations are evaluated in parallel, and each computation may spawn a future replacing itself with two new computations. An important property of assigning order identifiers is that all computations transitively spawned by the future computation N_f are also ordered before the continuation N_c. As explained above, if the spawning computation N has order identifier $(\mathtt{r}, \mathtt{d})$, the semantics assigns future N_f order identifier $(\mathtt{r}, \mathtt{d}+\mathtt{1})$ and continuation N_c order identifier $(\mathtt{r}+\mathtt{0.5}^{\mathtt{d}}, \mathtt{d}+\mathtt{1})$. Suppose computation N_f spawns another future, $N_{f'}$ with continuation $N_{c'}$. The semantics assigns $N_{f'}$ order identifier $(\mathtt{r}, \mathtt{d}+\mathtt{2})$ and $N_{c'}$ order identifier $(\mathtt{r}+\mathtt{0.5}^{\mathtt{d+1}}, \mathtt{d}+\mathtt{2})$. Note the following relation holds: $(\mathtt{r} < (\mathtt{r}+\mathtt{0.5}^{\mathtt{d+1}}) < (\mathtt{r}+\mathtt{0.5}^{\mathtt{d}}))$. Thus $N_{f'}$ is ordered before $N_{c'}$ which is ordered before N_c. All computations transitively spawned by N_c will be given order identifiers $\mathtt{r}'$ such that $(\mathtt{r}' \geq \mathtt{r}+\mathtt{0.5}^{\mathtt{d}})$ and will therefore be ordered after $N_{f'}$ and $N_{c'}$. It is straightforward to see that the demonstrated relation between order identifiers holds for all futures and their continuations.

3.1 Semantics

The operational semantics (see Figure 6) is defined by function F_{sd} from program M to a result R (where R is the same as it was in the sequential semantics). The transition rule $S \Rightarrow_{sd} S'$ maps a program state to a new program state. A program state is a process S which represents a collection of concurrently evaluating runtime terms (i.e. computations). Each computation maintains a local term context which is a three-tuple consisting of the placeholder p whose value is being computed by the term, the order identifier $(\mathtt{r}', \mathtt{d}')$ of the computation that spawned the future, and the computation's own order identifier, $(\mathtt{r}, \mathtt{d})$. The original program term is the only computation that is not a future. It is evaluated with term context $\langle \mathtt{main}, (-1,-1), (0,0)\rangle$, where $\mathtt{main}$ is a special placeholder value and $(-1,-1)$ signifies that it has no spawning parent.

Any references to a future's result in its continuation are replaced with a new placeholder variable. The semantics guarantees safety by preventing unsafe evaluation of the continuation beyond a $\varepsilon^{\bullet p}$ evaluation context. The evaluation context signifies that the term being evaluated in the hole is a continuation of the future corresponding to placeholder p. Evaluation of terms with $V^{\bullet p}$, $\mathtt{raise}\ X^{\bullet p}$, and $\mathtt{exit}\ V^{\bullet p}$ are restricted. For example, the term $(\mathtt{try}\ V^{\bullet p}\ \mathtt{handle}\ X \mapsto V')$ is stuck and cannot discard the handler because the future corresponding to p, which was spawned inside of the try body, has yet to complete and may require the handler defined by the try statement.

Runtime terms in a continuation may contain placeholder variables. The introduction of placeholder variables is discussed below as part of the *future* rule. In certain cases the result of a placeholder is required to proceed with evaluation and in other cases it is not. For example, in the term $(\mathtt{let}\ x = (p_1\ p_2)\ \mathtt{in}\ N)$ the abstraction result of p_1 is required for evaluation to proceed, but the result of argument p_2 is not required. The placeholder can simply be substituted into the λ-expression's body. Given a run-time term N with placeholder variables, the function $\mathcal{R}(N)$ annotates each placeholder variable p whose result is required with a $^+$ superscript. This distinction forces the continuation to perform a touch operation only on placeholder variables whose values are necessary to its evaluation. To guarantee that the program evaluates to a non-placeholder value, the program M is transformed to ($\texttt{let}\ x = M\ \texttt{in}\ \texttt{exit}\ x$). The exit statement forces a touch operation on variable x. This is necessary in the case that during evaluation x is replaced by a placeholder variable (e.g. if M is ($\texttt{let}\ x = \texttt{future}(M')\ \texttt{in}\ x$)).

Rule *seq* states that if $N \in S$ and $N \rightarrow_{seq} N'$ then $S \Rightarrow_{sd} S'$ where in the new process state S', term N is replaced by $\mathcal{R}(N')$. Since variables may be substituted by placeholders (e.g. under *apply* rule of $\rightarrow_{seq}$), the $\mathcal{R}$ function is applied to the new term. Note that all rules, except for the *seq* rule, are on terms that contain a $^{\bullet p}$ or a placeholder variable, and that both $^{\bullet p}$ and placeholders are introduced by the *future* rule (explained below). This implies that evaluation under $\rightarrow_{seq}$ and $\Rightarrow_{sd}$ are trivially equivalent in the absence of futures. The *let* rule allows continuations to proceed past the `let` evaluation context. The $\varepsilon^{\bullet p}$ evaluation context is only meant to disallow unsafe evaluation (e.g. discarding of try statements).

Rule *future* defines evaluation of a future-spawning term. Given term ($\texttt{let}\ x = \texttt{future}(M)\ \texttt{in}\ M') \in S$, the term is replaced in the process state with two new terms-one to evaluate the future computation M and one to evaluate the continuation M'. The continuation is evaluated in the evaluation context of the spawning term which includes any `try` statements that contain the spawning term. References to the variable x in M' are replaced by a fresh placeholder p' and function $\mathcal{R}$ replaces placeholders that need to be touched with p'^{+}. If the term context of the spawning computation is $\langle p, (\mathtt{r}, \mathtt{d}), (\mathtt{r}', \mathtt{d}')\rangle$, the term context of future computation is $\langle p', (\mathtt{r}', \mathtt{d}'), (\mathtt{r}', \mathtt{d}' + 1)\rangle$ and the term context of continuation is $\langle p, (\mathtt{r}, \mathtt{d}), (\mathtt{r}' + 0.5^{\mathtt{d}'}, \mathtt{d}' + 1)\rangle$. When the continuation requires the value of p'^{+} the semantics will know which computation to synchronize with based on the term context of future computation (via the *touch* rule). The continuation term is evaluated in the $\varepsilon^{\bullet p'}$ context so as to block the continuation from discarding a handler that may be required by the future corresponding to p'. When the future evaluates to a value, it removes the blocking context from its continuation (via rule *unblock*). For a computation to evaluate to a value (rather than to $V^{\bullet p''}$, for example) all of its futures must remove their corresponding blocking contexts.

The *raise* rule defines what happens when a future raises an escaping exception. Terms in S' represent valid computations and terms in S_c represent computations that have been invalidated by this future's `raise`. All computations that have been spawned as a result of evaluating the continuation of the raising future are invalid. All other computations are valid. Invalidated computations are replaced with $\perp$. The term $\varepsilon^{\bullet p}$ in the continuation of the future corresponding to p is replaced with a `raise` of the exception from the future, propagating the future's raise to the context where it was spawned. By replacing the term and the $^{\bullet p}$, the semantics ensures that the future's raise will never be propagated again. The *exit* rule requires that all computations that are logically ordered before the exiting computation have evaluated to values and therefore cannot invalidate the exiting computation.

$$
\begin{aligned}
N \in \mathit{RTTerms} &::= \ldots \mid \texttt{let}\ x = \texttt{future}(N)\ \texttt{in}\ N \mid \bot \mid p^{+} \mid N^{\bullet p}\\
V \in \mathit{Value} &::= \ldots \mid p\\
p \in \mathit{PhVars} &::= \{\texttt{main}, p_1, p_2, \ldots\}\\
C \in \mathit{TermContext} &::= \langle p \times (\mathit{real} \times \mathit{int}) \times (\mathit{real} \times \mathit{int})\rangle\\
S &::= \{(N_1)_{C_1}, \ldots, (N_n)_{C_n}\}\\
S|N_C &::= S \cup \{N_C\}
\end{aligned}
$$

$$
\mathcal{R}(N) = \begin{cases}
V & N = V\\
x & N = x\\
\texttt{exit}\ p^{+} & N = \texttt{exit}\ p\\
\texttt{raise}\ p^{+} & N = \texttt{raise}\ p\\
\texttt{try}\ \mathcal{R}(N')\ \texttt{handle}\ X \mapsto \mathcal{R}(N_h) & N = \texttt{try}\ N'\ \texttt{handle}\ X \mapsto N_h\\
\texttt{let}\ x = \texttt{if}\ p^{+}\ \texttt{then}\ \mathcal{R}(N_1)\ \texttt{else}\ \mathcal{R}(N_2) & N = (\texttt{let}\ x = \texttt{if}\ p\ \texttt{then}\ N_1\ \texttt{else}\ N_2\\
\qquad \texttt{in}\ \mathcal{R}(N) & \qquad \texttt{in}\ N)\\
\ldots &
\end{cases}
$$

$$
F_{sd}(M) = \begin{cases}
V & \texttt{if}\quad \{(\texttt{let}\ x = M\ \texttt{in}\ \texttt{exit}\ x)_{\langle \texttt{main},(-1,-1),(0,0)\rangle}\} \Rightarrow^{*}_{sd}\\
 & \qquad \{(N_1)_{C_1}, \ldots, (N_n)_{C_n}, (V)_{\langle \texttt{main},(-1,-1),(r,d)\rangle}\}\\
\texttt{error} & \texttt{otherwise}
\end{cases}
$$

EVALUATION RULES:

$$
\begin{aligned}
\varepsilon \in \mathit{EvalCntxt} ::= {} & [\,] \mid \texttt{try}\ \varepsilon\ \texttt{handle}\ X \mapsto N \mid \texttt{let}\ x = \varepsilon\ \texttt{in}\ N \mid \texttt{exit}\ \varepsilon \mid \texttt{raise}\ \varepsilon\\
& \mid \texttt{let}\ x = \texttt{if}\ \varepsilon\ \texttt{then}\ N_t\ \texttt{else}\ N_f\ \texttt{in}\ N \mid \texttt{let}\ x = (\varepsilon\ V)\ \texttt{in}\ N \mid \varepsilon^{\bullet p}
\end{aligned}
$$

$$
\frac{N \rightarrow_{seq} N'}{S|(\varepsilon[N])_C \Rightarrow_{sd} S|(\varepsilon[\mathcal{R}(N')])_C} \qquad (\mathit{seq})
$$

$$
\frac{\texttt{let}\ x = N\ \texttt{in}\ N' \rightarrow_{seq} N''}{S|(\varepsilon[\texttt{let}\ x = N^{\bullet p_1 \cdots \bullet p_n}\ \texttt{in}\ N'])_C \Rightarrow_{sd} S|(\varepsilon[N''^{\bullet p_1 \cdots \bullet p_n}])_C} \qquad (\mathit{let})
$$

$$
\frac{\begin{array}{c} C = \langle p, (r,d), (r',d')\rangle \quad p'\ \mathit{fresh}\\ C_f = \langle p', (r',d'), (r',d'+1)\rangle \quad C_c = \langle p, (r,d), (r'+0.5^{d'}, d'+1)\rangle \end{array}}{S|(\varepsilon[\texttt{let}\ x = \texttt{future}(N)\ \texttt{in}\ N'])_C \Rightarrow_{sd} S|(N)_{C_f}|(\varepsilon[\mathcal{R}(N'[x/p'])^{\bullet p'}])_{C_c}} \qquad (\mathit{future})
$$

$$
\frac{(V)_{\langle p,(r,d),(r',d')\rangle} \in S}{S|(\varepsilon[p^{+}])_C \Rightarrow_{sd} S|(\varepsilon[V])_C} \qquad \frac{(V)_{\langle p,(r,d),(r',d')\rangle} \in S}{S|(\varepsilon[N^{\bullet p}])_C \Rightarrow_{sd} S|(\varepsilon[N])_C} \qquad (\mathit{touch}) \quad (\mathit{unblock})
$$

$$
\frac{\begin{array}{c} C_f = \langle p, (r,d), (r',d')\rangle\\ S' = \{(N')_{\langle p_i,(r_i,d_i),(r'_i,d'_i)\rangle} \mid (N')_{\langle p_i,(r_i,d_i),(r'_i,d'_i)\rangle} \in S,\ (r'_i < r'\ \texttt{or}\ r'_i \geq (r + 0.5^{d-1}))\}\\ S_c = \{(\bot)_C \mid (N')_C \in S, (N')_C \notin S'\} \quad S'' = S' \cup S_c \end{array}}{S|(\texttt{raise}\ X)_{C_f}|(\varepsilon[N^{\bullet p}])_{C_c} \Rightarrow_{sd} S''|(\texttt{raise}\ X)_{C_f}|(\varepsilon[\texttt{raise}\ X])_{C_c}} \qquad (\mathit{raise})
$$

$$
\frac{\begin{array}{c} C = \langle p, (r',d'), (r,d)\rangle\\ (N_i)_{\langle p_i,(r'_i,d'_i),(r_i,d_i)\rangle} \notin S \quad r_i < r \quad N_i \neq V' \end{array}}{S|(\varepsilon[\texttt{exit}\ V])_C \Rightarrow_{sd} \{(V)_{\langle \texttt{main},(-1,-1),(r,d)\rangle}\}} \qquad (\mathit{exit})
$$

FIG. 6. SAFE DYNAMIC EVALUATION

3.2 Example

Consider the following program:

```
1.  let $x = $ future($M_1$) in
2.    try let $y = $ future($M_2$) in
3.          let $z = $ future($M_3$) in $M_4$
4.    handle $X \mapsto c$
```

Evaluation begins with a single term in the process state evaluating the above `let` expression. The program spawns three futures M_1, M_2 and M_3 resulting in a process state with four terms. Runtime terms N_1, N_2 and N_3 correspond to program terms M_1, M_2 and M_3, respectively. The continuation of these futures is the following runtime term in the process state: $(\texttt{try}\ N_4'^{\bullet p_3 \bullet p_2}\ \texttt{handle}\ X \mapsto c)^{\bullet p_1}$, where term N_4' corresponds to the evaluation of program term M_4. The bulleted evaluation context on term N_4' prevents the continuation from discarding the handler, because the futures corresponding to p_2 and p_3 (i.e. N_2 and N_3) may require it.

Consider what happens if term N_2 raises exception X without handling it internally. The $raise$ rule will propagate the exception into its continuation and invalidate all futures spawned by its continuation (i.e. N_3). N_3 is replaced with $\bot$, and the term $N_4'^{\bullet p_3 \bullet p_2}$ is replaced with the raise of exception X, resulting in term
$(\texttt{try raise}\ X\ \texttt{handle}\ X \mapsto c)^{\bullet p_1}$.

The continuation is now free to use the handler and evaluate to c. Note that the continuation still has the blocking context for p_1 from the first future preventing it from completing with value c. This is because N_1 may exit the program or raise an escaping exception invalidating its continuation's computation. Once N_1 evaluates to a value it notifies the continuation by removing the p_1 blocking context, allowing the main thread to complete with value c.

3.3 Semantic Equivalence

In this section we provide a proof sketch proving that the result of evaluating program M under safe dynamic semantics is the same as evaluating M under sequential semantics. For the proof, we define a transform function $\mathcal{T}$ to map a process state S in the safe dynamic semantics to a runtime term N in the sequential semantics. We use the transform function to show that for any process state S, if $S \Rightarrow_{sd}^{*} V$ then $\mathcal{T}(S) \Rightarrow_{seq}^{*} V$.

Given a process state S, the transform function $\mathcal{T}$ first identifies the computation N_f in the process state which is logically ordered before all other computations (i.e. the computation with the smallest `r` as its order identifier). If the continuation of N_f has not spawned any new computations, then the transform function will combine the future computation and its continuation to build runtime term $(\texttt{let}\ x = \texttt{future}(N_f)\ \texttt{in}\ N)$, where N is the continuation of N_f. If the continuation has split its computation by spawning futures, which also may have spawned other futures, then the transform function is recursively applied to all computations that have been spawned as a result of evaluating N_f's continuation to construct N_f's continuation term. The computations' order identifiers are used to identify which computations have been spawned from a given continuation. The transform function also replaces runtime terms that may appear in the safe dynamic semantics but not in the sequential semantics with equivalent terms. For example, invalidated computations represented as $\perp$ in the sequential semantics are replaced with $(\texttt{exit} \ -1)$ which is safe because those computations will never be reached in the evaluation of the runtime term due to an exception raise. The transform function and proof details are presented in an accompanying technical report [8].

Lemma 1. *If S is a final state that evaluates to V then $\mathcal{T}(S) \Rightarrow^*_{seq} V$.*
The lemma states that if a final process state S reached by evaluation under safe dynamic semantics evaluates to V, then the transform of the process state evaluates to V under sequential semantics.

Lemma 2. *If $S \Rightarrow_{sd} S'$ and $\mathcal{T}(S) \Rightarrow^*_{seq} N'$, then $\mathcal{T}(S') \Rightarrow^*_{seq} N'$.*
The lemma states that given an evaluation rule $S \Rightarrow_{sd} S'$, if the transform of S (i.e. $\mathcal{T}(S)$) yields runtime term N_s and the transform of S' (i.e. $\mathcal{T}(S')$) yields runtime term $N_{s'}$ then there exists a sequence of $\Rightarrow_{seq}$ rules from N_s, and a sequence of $\Rightarrow_{seq}$ rules from $N_{s'}$ that result in a common term N'. The lemma is proved by a case analysis on evaluation derivations $S \Rightarrow_{sd} S'$.

Theorem 1. *If $F_{sd}(M) = R$, then $F_{seq}(M) = R$.*
The result of evaluating program M under the safe dynamic semantics is guaranteed to be the same as the result of evaluating M under the sequential semantics. The proof is by induction on the length of $\Rightarrow_{sd}$ evaluation sequences. The base case is demonstrated by instantiating Lemma 1 and 2 and the inductive case is demonstrated by instantiating the inductive hypothesis and Lemma 2.

4 Instrumented Evaluation

The operational semantics defined thus far prevents a continuation from executing past a `try` expression if a future spawned within the try block has yet to complete. In this section, we present a flow-sensitive static analysis, program instrumentation, and a refined operational semantics that extracts more parallelism than this conservative treatment while still guaranteeing determinism. Informally, our solution is based on the observation that if a future reaches a point in its execution where it will no longer raise escaping exception X, then its continuation can proceed past a handler for exception X.

In the instrumented semantics, the blocking evaluation context is of the form $\varepsilon^{\bullet(p,\Sigma)}$. The evaluation context signifies that the term being evaluated in the whole is a continuation of the future that corresponds to placeholder p and that the evaluation of the future may result in a raise of escaping exception $X \in \Sigma$ or may exit the program if $\texttt{exit} \in \Sigma$. A continuation evaluating runtime term $(\texttt{try}\ V^{\bullet(p,\Sigma)}\ \texttt{handle}\ X \mapsto N_h)$, where $X \notin \Sigma$ may proceed past the `try` expression (unlike in the previously presented semantics), thus discarding the handler. Static instrumentation specifies which escaping exceptions an evaluating future computation may raise and whether or not it may exit the program. We present an operational semantics that leverages the instrumentation so that a future computation notifies its continuation immediately when its computation and the futures it creates may no longer raise an escaping exception or perform an exit operation (by removing elements from Σ in the blocking context of its continuation).

4.1 Static Analysis and Program Instrumentation

Our instrumentation assumes the presence of control-flow analysis $Flow_P(x)$ [6, 7] which maps variable x to all possible values it may be bound to during the evaluation of program P. Program term M is instrumented with a *grant* set Σ (represented by superscript $\triangleright\Sigma$) and a *nogrant* set Σ' (represented by subscript $\triangleleft\Sigma'$). The *grant* set Σ includes all escaping exceptions that may be raised by the instrumented term, and a special `exit` element if the term may exit the program. The *nogrant* set Σ' includes all escaping exceptions that may be raised after evaluation of the instrumented term by the enclosing term and the `exit` element if the enclosing term may exit the program after evaluation of the instrumented term. The *nogrant* set ensures a computation does not prematurely notify its continuation that it cannot reach an escaping exception or exit. Term M is transformed to the instrumented term $\widetilde{T}$ defined by the following grammar:

$$\begin{aligned} T \in InstTerms' &::= V_l \mid x_l \mid \texttt{exit}_l\, x \mid \texttt{raise}_l\, x \\ &\mid \texttt{try}_l\, \widetilde{T}\, \texttt{handle}\, X \mapsto \widetilde{T} \mid \ldots \\ \widetilde{T} \in InstTerms &::= T^{\triangleright\Sigma}_{\triangleleft\Sigma'} \end{aligned}$$

When a future f is spawned its continuation is evaluated in context: $\varepsilon^{\bullet(p,\Sigma)}$, where Σ is initially equal to f's *grant* set. Let f be the following future computation:

$$(\texttt{let}\; x = \texttt{if}\; y\; \texttt{then}\; \texttt{raise}\; z\; \texttt{else}\; M_f\; \texttt{in}\; M)$$

If $Flow_P(z) = \{X\}$ and computations M_f and M do not raise an escaping exception X (or exit), then the continuation of f may discard X's handler as soon as control enters the false branch during the evaluation of f. Static instrumentation allows the instrumented semantics to notify f's continuation when control enters the false branch. In the instrumented term below Σ and Σ_f are the *grant* sets (i.e. the sets of possible escaping exceptions and exit that may be reached) of M and M_f, respectively.

$$(\texttt{let}\; x = \texttt{if}\; y\; \texttt{then}\; (\texttt{raise}\; z)^{\triangleright\{X\}}_{\triangleleft\Sigma}\; \texttt{else}\; {M_f}^{\triangleright\Sigma_f}_{\triangleleft\Sigma}\; \texttt{in}\; M^{\triangleright\Sigma}_{\triangleleft\{\}})^{\triangleright\{X\}\cup\Sigma_f\cup\Sigma}_{\triangleleft\{\}}$$

As mentioned above M_f and M do not raise X (i.e. $X \notin (\Sigma_f \cup \Sigma)$). The *grant* set of the entire term captures all escaping exceptions that may be raised by the future (i.e. $\{X\} \cup \Sigma_f \cup \Sigma$). The *nogrant* set is empty; the term represents the entire future computation and therefore has no enclosing term. If f computes placeholder p, the continuation of f evaluates in context $\varepsilon^{\bullet(p,\Sigma')}$, where $\Sigma' = (\{X\} \cup \Sigma_f \cup \Sigma)$. When f's evaluation enters the false branch of the `if-then-else` statement, f will remove those escaping exceptions it can no longer reach from Σ' in the evaluation context of its continuation. Since both the false branch and the body of the statement do not raise X, the future will remove element X from Σ'. The result is that f's continuation will evaluate in the following evaluation context: $\varepsilon^{\bullet(p,\Sigma_f\cup\Sigma)}$.

Relation $\mathcal{I}(\widetilde{T})$ defines constraints on instrumented terms $\widetilde{T}$ (see Figure 7). Variable and value terms are uniquely labeled with their static location, and each term has its own *nogrant* set depending on its context. Of course, value and variable occurrences may not raise exceptions or exit the program so their *grant* sets are always empty. An exit statement obviously exits and therefore has *grant* set $\{\texttt{exit}\}$, and a raise statement clearly raises an exception. Since exceptions are first-class, the raise statement's *grant* set contains all exceptions it *may* raise (i.e. the *grant* set for term $(\texttt{raise}_l\ x)$ is $Flow_P(x)$). A `try` expression's *grant* set includes all exceptions (and `exit`) that escape from the `try` block except the handled exception, and all escaping exceptions from the handler block. Thus, the continuation of a future f will not be forced to wait for a grant on an exception that is handled internally by f. The *nogrant* set for the `try` expression's try block includes the try expression's *nogrant* set, all escaping exceptions raised by the handler block, and the handled exception. Computing the *grant* and *nogrant* sets for the `try`'s handler block is straightforward, as is the case for value binding let-expressions and `if-then-else` expressions.

Since abstractions are first-class, the *grant* set of an application term with abstraction variable y is the union of *grant* sets for $\widetilde{T}_i$ where $\lambda\, z.\widetilde{T}_i \in Flow_P(y)$ and the *grant* set for the body of the `let`-expression. An abstractions may appear in different contexts; therefore, the body of a λ-expression must be instrumented with a conservative approximation for its *nogrant* set. The *nogrant* set is the union of *nogrant* sets for each context the abstraction may be applied. This is demonstrated in the instrumentation constraints presented in Figure 7 by requiring that the *nogrant* sets associated with the bodies of each potential abstraction is a *subset* of the set of exceptions for the current context. This overly conservative *nogrant* set disallows grants that are safe. The disallowed grants that should have been granted during evaluation of the application are applied at runtime after evaluating the application (see Figure 8). The *grant* set for a term that spawns a future consists of the *grant* set of the future term and the continuation. Thus if the term itself is spawned as a future, its continuation will need to wait for both the (sub) future and the original future to grant on exceptions and `exit`. A future computation has an empty *nogrant* set because it is evaluated as a separate computation.

4.2 Semantics

Terms instrumented with *grant* and *nogrant* sets are evaluated using the semantics defined in Figures 8 and 9. In Figure 8 we omit instrumentation that is not relevant to evaluation. A local evaluation rule $\widetilde{N} \rightarrow_{is} \langle \widetilde{N}', \varSigma \rangle$ reduces an instrumented runtime term $\widetilde{N}$ to a new instrumented runtime term $\widetilde{N}'$ and a grant effect $\varSigma$. The grant effect represents the escaping exceptions (and `exit`) that were reachable by $\widetilde{N}$ but not reachable by $\widetilde{N}'$.

For the $\overrightarrow{try}$ rule, the try block evaluates to a value and thus does not require the exception handler. The $\rightarrow_{is}$ evaluator will compute a grant effect consisting of those elements in the handler's *grant* set that are not in its *nogrant* set. If the body of the `try` statement raises the handled exception (i.e. rule $\overrightarrow{handle}$), the $\rightarrow_{is}$ evaluator grants exceptions in the raise statement's *grant* set (i.e. the static approximation of which exceptions *may* have been raised by this statement), that are not in its *nogrant* set. Note that the instrumentation constraints ensure the exception being handled, which is clearly in the *grant* set, is also in the *nogrant* set disallowing the rule to grant on the raised exception. This is correct because the instrumentation constraints ensure that a continuation of a future does not wait for exceptions internally handled by its future. If another exception is raised in the try block (i.e. $\overrightarrow{tryraise}$ rule), the exception is propagated and

$$\frac{}{\mathcal{I}(V_{l\,\triangleleft\Sigma}^{\triangleright\{\}})} \qquad \frac{}{\mathcal{I}(x_{l\,\triangleleft\Sigma}^{\triangleright\{\}})} \qquad \frac{}{\mathcal{I}((\texttt{exit}_l\ x)_{\triangleleft\Sigma}^{\triangleright\{\texttt{exit}\}})} \qquad \frac{\Sigma = \mathit{Flow}_P(x)}{\mathcal{I}((\texttt{raise}_l\ x)_{\triangleleft\Sigma'}^{\triangleright\Sigma})}$$

$$\frac{\mathcal{I}(T^{\triangleright\Sigma}_{\triangleleft\Sigma'\cup\Sigma_h\cup\{X\}}) \quad \mathcal{I}({T_h}^{\triangleright\Sigma_h}_{\triangleleft\Sigma'})}{\mathcal{I}((\texttt{try}_l\ T^{\triangleright\Sigma}_{\triangleleft\Sigma'\cup\Sigma_h\cup\{X\}}\ \texttt{handle}\ X \mapsto {T_h}^{\triangleright\Sigma_h}_{\triangleleft\Sigma'})^{\triangleright(\Sigma\setminus\{X\})\cup\Sigma_h}_{\triangleleft\Sigma'})}$$

$$\frac{\mathcal{I}(T^{\triangleright\Sigma}_{\triangleleft\Sigma'})}{\mathcal{I}(\texttt{let}\ x = V\ \texttt{in}\ T^{\triangleright\Sigma}_{\triangleleft\Sigma'})^{\triangleright\Sigma}_{\triangleleft\Sigma'}}$$

$$\frac{\mathcal{I}({T_t}^{\triangleright\Sigma_t}_{\triangleleft\Sigma''\cup\Sigma}) \quad \mathcal{I}({T_f}^{\triangleright\Sigma_f}_{\triangleleft\Sigma''\cup\Sigma}) \quad \mathcal{I}(T^{\triangleright\Sigma}_{\triangleleft\Sigma''})}{\mathcal{I}((\texttt{let}\ x = \texttt{if}\ y\ \texttt{then}\ {T_t}^{\triangleright\Sigma_t}_{\triangleleft\Sigma''\cup\Sigma}\ \texttt{else}\ {T_f}^{\triangleright\Sigma_f}_{\triangleleft\Sigma''\cup\Sigma}\ \texttt{in}\ T^{\triangleright\Sigma}_{\triangleleft\Sigma''})^{\triangleright\Sigma_t\cup\Sigma_f\cup\Sigma}_{\triangleleft\Sigma''})}$$

$$\frac{\begin{array}{c}\mathit{Flow}_P(y) = \{\lambda\, z_1.T_1, \ldots, \lambda\, z_n.T_n\} \quad \mathcal{I}({T_1}^{\triangleright\Sigma_1}_{\triangleleft\Sigma'_1}), \ldots, \mathcal{I}({T_n}^{\triangleright\Sigma_n}_{\triangleleft\Sigma'_n}) \\ \mathcal{I}(T^{\triangleright\Sigma}_{\triangleleft\Sigma'}) \quad \Sigma'_i \subseteq (\Sigma \cup \Sigma') \quad \Sigma'' = \bigcup_{(i=1)}^{n} \Sigma_i\end{array}}{\mathcal{I}((\texttt{let}\ x = (y\ z)\ \texttt{in}\ T^{\triangleright\Sigma}_{\triangleleft\Sigma'})^{\triangleright\Sigma\cup\Sigma''}_{\triangleleft\Sigma'})}$$

$$\frac{\mathcal{I}({T_f}^{\triangleright\Sigma_f}_{\triangleleft\{\}}) \quad \mathcal{I}(T^{\triangleright\Sigma}_{\triangleleft\Sigma'})}{\mathcal{I}((\texttt{let}\ x = \texttt{future}({T_f}^{\triangleright\Sigma_f}_{\triangleleft\{\}})\ \texttt{in}\ T^{\triangleright\Sigma}_{\triangleleft\Sigma'})^{\triangleright\Sigma_f\cup\Sigma}_{\triangleleft\Sigma'})}$$

Fig. 7. Instrumentation Constraints

since the handler is not invoked, the $\rightarrow_{is}$ evaluator will grant elements in the handler's *grant* set that are not in its *nogrant* set.

The grant effect computed by rule $\overrightarrow{bind}$ for a value-binding `let`-expression is empty, because the new runtime term may raise the same set of escaping exceptions as the reduced term. If the expression being bound results in a raise of an exception, rule $\overrightarrow{bindraise}$ will compute a grant effect that includes escaping exceptions that may be raised by the body of the `let`-expression, which will never be reached, (i.e. Σ) as long as those exceptions may not be raised by the raise statement (i.e. Σ_r) or by the computation following the entire term (i.e. Σ'). The rule also computes the new *nogrant* set that results from propagating the raise without evaluating the `let`-expression body. The new *nogrant* set is equal to the *nogrant* set of the `let`-expression body.

The $\overrightarrow{if}$ rule computes the grant effect resulting from taking a branch of an `if-then-else` statement, and recomputes the *grant* set of the entire term based on the branch taken. If the true (false) branch is taken, the *grant* set of the entire term is the union of the true (false) branch's *grant* set and the *grant* set of the `let`-body. The grant effect consists of exceptions raised (and `exit`) by the false (true) branch that cannot be raised by the true (false) branch, the `let`-body, or the computation that follows.

Rule $\overrightarrow{apply}$ computes the grant effect for an application term. The *nogrant* set of the abstraction body (i.e. Σ'_1) may be overly conservative, not allowing the evaluation of the body to grant on certain exceptions. Thus the runtime will grant all exceptions in the *nogrant* set of the abstraction body that are not raised by the rest of the term (i.e. $\Sigma'_1 \setminus (\Sigma \cup \Sigma')$) as soon as the application has completed evaluation. This is achieved by replacing the body of the `let`-expression with a grant effect causing the grant effect to be applied before evaluating the body. The grant effect immediately computed by the rule consists of those exceptions in the `let`-expression's *grant* set modulo those in the *grant* set of the abstraction's body (i.e. Σ_1), those that are reachable from the `let`-body (i.e. Σ) and those associated with the computation following the `let`-body (i.e. Σ'). Note that the grant effect will include exceptions added to the *grant* set based on the static approximation of which abstractions *may* have been bound to V as long as the exceptions may not be raised by the rest of the term or by the abstraction value actually bound to V at runtime.

The global evaluation rules are mostly analogous to the evaluation rules for the safe dynamic semantics. The $unblock$ rule is worth noting because it removes the blocking evaluation for future f from a continuation as soon as f reaches a point where it has granted everything that was in its *grant* set. This allows a future computation to evaluate to a value (rather than a blocked value) before its futures complete, if they are guaranteed to not invalidate its evaluation by exiting or raising an escaping exception. Thus unlike the safe dynamic semantics, in the instrumented semantics a continuation touching a placeholder corresponding to future f does not need to block until all of f's futures complete. Three new rules are also defined: $grantmain$, $grant$ and try. The first two deal with grant effects and the try allows computation within a continuation to discard an exception handler if its future indicates it is safe to do so.

The $grantmain$ rule ignores grant effects from the main computation, because it is not a future of any continuation. When a local evaluation reduces to a term and grant effect $\langle \widetilde{N}, \Sigma \rangle$ the $grant$ rule will grant elements in Σ that are safe to grant. An element is safe to grant if the granting computation cannot reach the element (i.e. it is not in Σ_f) and the granting future is not a continuation of another future that still may reach the element (i.e. it is not in Σ''). To compute Σ'', we use function $\mathcal{A}_\bullet$, which computes the set of futures a given term is a continuation of. The grant action is reflected in the $grant$ rule by removing elements Σ_g from Σ' in the continuation's $\varepsilon^{\bullet(p,\Sigma')}$ context. The grant effect is then propagated to the continuation which may itself be a future.

The try rule exploits concurrency that could not be availed in the absence of instrumentation. A continuation may proceed past a try statement before its futures complete if the futures and all the futures they spawn will not require the handler defined by the `try` expression. This rule is similar to the let rule which allows evaluation of a continuation to proceed past a `let` term, except the try rule is conditional on the blocking instrumentation indicating it is safe to discard the handler.

4.3 Example

The following example shows how the instrumented semantics allows for greater parallelism than the safe dynamic semantics. For brevity we have omitted the instrumentation from the example, but we assume the program has been instrumented to satisfy the instrumentation constraints presented in Figure 7. We explain in the text any instru-

$$
\begin{aligned}
N \in RTTerms' &::= V \mid \texttt{exit}\ V \mid \texttt{raise}\ X \mid \texttt{try}\ \widetilde{N}\ \texttt{handle}\ X \mapsto \widetilde{N} \mid \texttt{let}\ x = \texttt{if}\ V\ \texttt{then}\ \widetilde{N}\ \texttt{else}\ \widetilde{N}\ \texttt{in}\ \widetilde{N} \mid \ldots \\
\widetilde{N} \in RTTerms &::= N^{\triangleright\Sigma}_{\triangleleft\Sigma} \mid \bot \mid p^{+} \mid \widetilde{N}^{\bullet(p,\Sigma)} \mid \langle \widetilde{N}, \Sigma \rangle \\
V \in Value &::= \ldots \mid p \\
p \in PhVars &::= \{\texttt{main}, p_1, p_2, \ldots\} \\
C \in TermContext &::= \langle p \times (real \times int) \times (real \times int) \rangle \\
S &::= \{\widetilde{(N_1)}_{C_1}, \ldots, \widetilde{(N_n)}_{C_n}\} \\
S|\widetilde{N}_C &::= S \cup \{\widetilde{N}_C\}
\end{aligned}
$$

$$
\mathcal{A}_{\bullet}(\widetilde{N}) \ = \ \begin{cases} \{(p,\Sigma)\} \cup \mathcal{A}_{\bullet}(\varepsilon[\widetilde{N}']) & \texttt{if}\ \widetilde{N} = \varepsilon[\widetilde{N}'^{\bullet(p,\Sigma)}] \\ \phi & \texttt{otherwise} \end{cases}
$$

$$
F_{is}(M) \ = \ \begin{cases} V & \texttt{if}\ \ \{(\texttt{let}\ x = \widetilde{T}\ \texttt{in}\ \texttt{exit}_l\ x)_{\langle \texttt{main},(-1,-1),(0,0)\rangle}\} \Rightarrow^{*}_{sd} \\ & \qquad \{(\widetilde{N}_1)_{C_1}, \ldots, (\widetilde{N}_n)_{C_n}, (V^{\triangleright\{\}}_{\triangleleft\{\}})_{\langle \texttt{main},(-1,-1),(r,d)\rangle}\} \\ & \qquad \texttt{where}\ \widetilde{T}\ \texttt{is instrumented version of}\ M \\ \texttt{error} & \texttt{otherwise} \end{cases}
$$

EVALUATION RULES:

$$
\begin{aligned}
\varepsilon \in EvalCntxt ::= [\,] \mid \texttt{try}\ \varepsilon\ \texttt{handle}\ X \mapsto \widetilde{N} \mid \texttt{let}\ x = \varepsilon\ \texttt{in}\ \widetilde{N} \mid \texttt{exit}\ \varepsilon \mid \texttt{raise}\ \varepsilon \\
\mid \ \texttt{let}\ x = \texttt{if}\ \varepsilon\ \texttt{then}\ \widetilde{N}_t\ \texttt{else}\ \widetilde{N}_f\ \texttt{in}\ \widetilde{N} \mid \texttt{let}\ x = (\varepsilon\ V)\ \texttt{in}\ \widetilde{N} \mid \varepsilon^{\bullet(p,\Sigma)}
\end{aligned}
$$

$$
\begin{array}{lll}
\texttt{try}\ V\ \texttt{handle}\ X \mapsto N^{\triangleright\Sigma_h}_{\triangleleft\Sigma'} & \rightarrow_{is} \langle V, \Sigma_h \backslash \Sigma' \rangle & (\overrightarrow{try}) \\[1ex]
\texttt{try}\ (\texttt{raise}\ X)^{\triangleright\Sigma_r}_{\triangleleft\Sigma'_r}\ \texttt{handle}\ X \mapsto \widetilde{N} & \rightarrow_{is} \langle \widetilde{N}, \Sigma_r \backslash \Sigma'_r \rangle & (\overrightarrow{handle}) \\[1ex]
\texttt{try}\ (\texttt{raise}\ X')^{\triangleright\Sigma_r}_{\triangleleft\Sigma'_r}\ \texttt{handle}\ X \mapsto N^{\triangleright\Sigma_h}_{\triangleleft\Sigma'} & \rightarrow_{is} \langle (\texttt{raise}\ X')^{\triangleright\Sigma_r}_{\triangleleft\Sigma'_r}, \Sigma_h \backslash \Sigma' \rangle & (\overrightarrow{tryraise}) \\[1ex]
\texttt{let}\ x = V\ \texttt{in}\ N^{\triangleright\Sigma}_{\triangleleft\Sigma'} & \rightarrow_{is} \langle (N[x/V])^{\triangleright\Sigma}_{\triangleleft\Sigma'}, \phi \rangle & (\overrightarrow{bind}) \\[1ex]
\texttt{let}\ x = (\texttt{raise}\ X)^{\triangleright\Sigma_r}_{\triangleleft\Sigma'_r}\ \texttt{in}\ N^{\triangleright\Sigma}_{\triangleleft\Sigma'} & \rightarrow_{is} \langle (\texttt{raise}\ X)^{\triangleright\Sigma_r}_{\triangleleft\Sigma'}, \Sigma \backslash (\Sigma' \cup \Sigma_r) \rangle & (\overrightarrow{bindraise})
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ x = \texttt{if}\ V\ \texttt{then}\ {N_t}^{\triangleright\Sigma_t}_{\triangleleft\Sigma''\cup\Sigma} \\
\qquad \texttt{else}\ {N_f}^{\triangleright\Sigma_f}_{\triangleleft\Sigma''\cup\Sigma}\ \texttt{in}\ N^{\triangleright\Sigma}_{\triangleleft\Sigma''}
\end{array}
\rightarrow_{is}
\begin{cases}
\langle (\texttt{let}\ x = {N_t}^{\triangleright\Sigma_t}_{\triangleleft\Sigma''\cup\Sigma}\ \texttt{in}\ N^{\triangleright\Sigma}_{\triangleleft\Sigma''})^{\triangleright\Sigma_t\cup\Sigma}_{\triangleleft\Sigma''}, & V = \texttt{true} \\
\quad \Sigma_f \backslash (\Sigma_t \cup \Sigma'' \cup \Sigma) \rangle & \\
\langle (\texttt{let}\ x = {N_f}^{\triangleright\Sigma_f}_{\triangleleft\Sigma''\cup\Sigma}\ \texttt{in}\ N^{\triangleright\Sigma}_{\triangleleft\Sigma''})^{\triangleright\Sigma_f\cup\Sigma}_{\triangleleft\Sigma''}, & V = \texttt{false} \\
\quad \Sigma_t \backslash (\Sigma_f \cup \Sigma'' \cup \Sigma) \rangle &
\end{cases}
\qquad (\overrightarrow{if})
$$

$$
\begin{array}{llll}
(\texttt{let}\ x = (V\ V')\ \texttt{in}\ N^{\triangleright\Sigma}_{\triangleleft\Sigma'})^{\triangleright\Sigma''}_{\triangleleft\Sigma'} & \rightarrow_{is} \langle (\texttt{let}\ x = {N'}^{\triangleright\Sigma_1}_{\triangleleft\Sigma'_1}[y/V'] & V = \lambda\, y.({N'}^{\triangleright\Sigma_1}_{\triangleleft\Sigma'_1}) & (\overrightarrow{apply}) \\
& \quad \texttt{in}\ \langle N^{\triangleright\Sigma}_{\triangleleft\Sigma'}, \Sigma'_1 \backslash (\Sigma \cup \Sigma') \rangle)^{\Sigma_1\cup\Sigma}_{\Sigma'},\ \Sigma'' \backslash (\Sigma_1 \cup \Sigma \cup \Sigma') \rangle & &
\end{array}
$$

FIG. 8. LOCAL EVALUATION RULES FOR INSTRUMENTED SEMANTICS

$$\frac{\widetilde{N} \rightarrow_{is} \langle \widetilde{N}', \Sigma \rangle}{S|(\varepsilon[\widetilde{N}])_C \Rightarrow_{is} S|(\varepsilon[\langle \mathcal{R}(\widetilde{N}'), \Sigma \rangle])_C} \qquad (local)$$

$$\frac{(\texttt{let}\ x = \widetilde{N}\ \texttt{in}\ N') \rightarrow_{is} \langle \widetilde{N}'', \Sigma \rangle}{S|(\varepsilon[\texttt{let}\ x = \widetilde{N}^{\bullet(p_1,\Sigma_1)\ldots\bullet(p_n,\Sigma_n)}\ \texttt{in}\ \widetilde{N}'])_C \Rightarrow_{sd} S|(\varepsilon[\langle \widetilde{N}''^{\bullet(p_1,\Sigma_1)\ldots\bullet(p_n,\Sigma_n)}, \Sigma \rangle])_C} \qquad (let)$$

$$\frac{\begin{array}{c}\texttt{t}', \texttt{t}''\ fresh \quad C = \langle p, (r,d), (r',d') \rangle \quad p'\ fresh \\ C_f = \langle p', (r',d'), (r',d'+1) \rangle \quad C_c = \langle p, (r,d), (r'+0.5^{d'}, d'+1) \rangle\end{array}}{S|(\varepsilon[\texttt{let}\ x = \texttt{future}(N^{\triangleright \Sigma_f}_{\triangleleft \{\}})\ \texttt{in}\ \widetilde{N}'])_C \Rightarrow_{sd} S|(N^{\triangleright \Sigma_f}_{\triangleleft \{\}})_{C_f}|(\varepsilon[\mathcal{R}(\widetilde{N}'[x/p'])^{\bullet(p',\Sigma_f)}])_{C_c}} \qquad (future)$$

$$\frac{(V^{\triangleright\{\}}_{\triangleleft\{\}})_{\langle p,(r,d),(r',d')\rangle} \in S}{S|(\varepsilon[p^+])_C \Rightarrow_{sd} S|(\varepsilon[V])_C} \qquad \frac{}{S|(\varepsilon[\widetilde{N}^{\bullet(p,\phi)}])_C \Rightarrow_{is} S|(\varepsilon[\widetilde{N}])_C} \qquad (touch) \quad (unblock)$$

$$\frac{\begin{array}{c}C_f = \langle p, (r,d), (r',d') \rangle \\ S' = \{(\widetilde{N}')_{\langle p_i,(r_i,d_i),(r'_i,d'_i)\rangle} \mid (\widetilde{N}')_{\langle p_i,(r_i,d_i),(r'_i,d'_i)\rangle} \in S,\ (r'_i < r'\ \texttt{or}\ r'_i \geq (r + 0.5^{d-1}))\} \\ S_c = \{(\bot)_C \mid (\widetilde{N}')_C \in S, (\widetilde{N}')_C \notin S'\} \quad S'' = S' \cup S_c\end{array}}{S|((\texttt{raise}\ X)^{\triangleright \Sigma_f}_{\triangleleft \Sigma'_f})_{C_f}|(\varepsilon[N^{\triangleright \Sigma_c \bullet(p,\Sigma)}_{\triangleleft \Sigma'_c}])_{C_c} \Rightarrow_{sd} S''|((\texttt{raise}\ X)^{\triangleright \Sigma_f}_{\triangleleft \Sigma'_f})_{C_f}|(\varepsilon[(\texttt{raise}\ X)^{\triangleright \Sigma_f}_{\triangleleft \Sigma'_c}])_{C_c}} \qquad (raise)$$

$$\frac{\begin{array}{c}C = \langle p, (r',d'), (r,d) \rangle \\ (\widetilde{N}_i)_{\langle p_i,(r'_i,d'_i),(r_i,d_i)\rangle} \notin S \quad r_i < r \quad \widetilde{N}_i \neq V'^{\triangleright\{\}}_{\triangleleft\{\}}\end{array}}{S|(\varepsilon[(\texttt{exit}\ V)^{\triangleright\{\texttt{exit}\}}_{\triangleleft \Sigma}])_C \Rightarrow_{sd} \{(V^{\triangleright\{\}}_{\triangleleft\{\}})_{\langle \texttt{main},(-1,-1),(r,d)\rangle}\}} \qquad (exit)$$

$$\frac{C = \langle \texttt{main}, (-1,-1), (r,d) \rangle}{S|(\varepsilon[\langle \widetilde{N}, \Sigma \rangle])_C \Rightarrow_{is} S|(\varepsilon[\widetilde{N}])_C} \qquad (grantmain)$$

$$\frac{\begin{array}{c}C_f = \langle p, (r,d), (r',d') \rangle \quad \mathcal{A}_\bullet(\varepsilon[N^{\triangleright \Sigma_f}_{\triangleleft \Sigma'_f}]) = \{(p_1,\Sigma_1), \ldots, (p_n,\Sigma_n)\} \\ \Sigma'' = \bigcup_{(i=1)}^n \Sigma_i \quad \Sigma_g = \Sigma \backslash (\Sigma'' \cup \Sigma_f)\end{array}}{S|(\varepsilon[\langle N^{\triangleright \Sigma_f}_{\triangleleft \Sigma'_f}, \Sigma \rangle])_{C_f}|(\varepsilon[\widetilde{N}_c^{\bullet(p,\Sigma')}])_{C_c} \Rightarrow_{is} S|(\varepsilon[N^{\triangleright \Sigma_f}_{\triangleleft \Sigma'_f}])_{C_f}|(\varepsilon[\langle \widetilde{N}_c^{\bullet(p,\Sigma' \backslash \Sigma_g)}, \Sigma_g \rangle])_{C_c}} \qquad (grant)$$

$$\frac{(\texttt{try}\ \widetilde{N}\ \texttt{handle}\ X \mapsto \widetilde{N}') \rightarrow_{is} \langle \widetilde{N}'', \Sigma' \rangle \quad \Sigma'' = \bigcup_{(i=1)}^n \Sigma_i \quad X \notin \Sigma''}{S|(\varepsilon[\texttt{try}\ \widetilde{N}^{\bullet(p_1,\Sigma_1)\ldots\bullet(p_n,\Sigma_n)}\ \texttt{handle}\ X \mapsto \widetilde{N}'])_C \Rightarrow_{is} S|(\varepsilon[\langle \widetilde{N}''^{\bullet(p_1,\Sigma_1)\ldots\bullet(p_n,\Sigma_n)}, \Sigma' \rangle])_C} \qquad (try)$$

Fig. 9. Global Evaluation Rules for Instrumented Semantics

mentation that is relevant to the evaluation of the program.

```
1.  let $x$ = future($\widetilde{T}_1$) in
2.    try let $y$ = future($\widetilde{T}_2$) in
3.          let $z$ = future(let $w$ = if false then raise $X$
4.                                     else $\widetilde{T}_3$ in $\widetilde{T}'_3$)
5.          in raise $X'$
6.    handle $X \mapsto c$
```

Let $\widetilde{N}_1$ and $\widetilde{N}_2$ be runtime terms in the process state corresponding to instrumented terms $\widetilde{T}_1$ and $\widetilde{T}_2$, respectively, $\widetilde{N}_3$ be the runtime term for the `if-then-else` expression on line 3, and $\widetilde{N}_4$ be the following runtime term:

$$(\texttt{try}\ (\texttt{raise}\ X')^{\bullet(p_3,\Sigma)\bullet(p_2,\Sigma_2)}\ \texttt{handle}\ X \mapsto c)^{\bullet(p_1,\Sigma_1)}$$

In the above runtime term, Σ is the *grant* set for the `if-then-else` expression, Σ_1 is the *grant* set for $\widetilde{N}_1$ and Σ_2 is the *grant* set for $\widetilde{N}_2$. Assume that $X \notin \Sigma_2$ (i.e. $\widetilde{T}_2$ may not raise escaping exception X). Since $X \in \Sigma$ due to the raise in the true branch of the future computation, the *try* rule does not hold for $\widetilde{N}_4$ and the handler cannot be discarded. Once control enters the false brach during the evaluation of $\widetilde{N}_3$, the $\rightarrow_{is}$ evaluator will compute a grant effect that includes elements in the *grant* set of the true branch (i.e. $\{X\}$) that are not in the *grant* and *nogrant* sets of the false branch. Let Σ_3 and Σ'_3 be the *grant* and *nogrant* sets of $\widetilde{T}_3$ and assume that $X \notin (\Sigma_3 \cup \Sigma'_3)$ (i.e. the false branch and the body of the `if` expression do not raise an escaping exception X). According to the $\overrightarrow{if}$ rule, the grant effect Σ_g contains X. The *grant* rule removes X from Σ of the blocking evaluation context associated with p_3 in term $\widetilde{N}_4$. The grant would be propagated but since $\widetilde{N}_4$ is not a future computation (i.e. its term context is $\langle \texttt{main}, (-1,-1), (r,d)\rangle$), the *grantmain* rule applies. Since X is no longer in Σ the *try* rule applies for term $\widetilde{N}_4$ allowing evaluation to proceed past the handler even though the future computations corresponding to p_2 and p_3 have yet to complete.

4.4 Semantic Equivalence

In this section we provide a proof sketch proving that evaluating program M under the instrumented semantics has the same result as evaluating M under the safe dynamic semantics. For the proof, we define a transform function $\mathcal{U}$ to map a process state S_i in the instrumented semantics to a process state S_s in the safe dynamic semantics. The transform function $\mathcal{U}$ simply erases instrumentation, replaces evaluation context $\bullet^{(p,\Sigma)}$ with $\bullet^{p}$ and replaces instrumented runtime term $\langle \widetilde{N}, \Sigma\rangle$ with runtime term N. We use the transform function $\mathcal{U}$ to prove that for state S_i, if $S_i \Rightarrow^*_{is} V$ then $\mathcal{U}(S_i) \Rightarrow^*_{sd} V$. The proof details are presented in an accompanying technical report [8].

Lemma 3. *If S_i is a final state with result R then $\mathcal{U}(S_i) \Rightarrow^*_{sd} S_s$, and S_s is a final state in the safe dynamic semantics with result R.*
The proof states that if the final process state S_i reached by evaluation under instrumented semantics results in R, then the transform of the process state evaluates under safe dynamic semantics to a final state S_s with result R.

Lemma 4. *If $S_i \Rightarrow_{is} S'_i$ and $\mathcal{U}(S_i) \Rightarrow^*_{sd} S_s$, then $\mathcal{U}(S'_i) \Rightarrow^*_{sd} S_s$.*
The lemma states that given an instrumented evaluation rule $S_i \Rightarrow_{is} S'_i$ the transform of S_i and S'_i are equivalent process states in the safe dynamic semantics. In other words, under $\Rightarrow_{sd}$ there exists a sequence of rules starting from $\mathcal{U}(S_i)$ and a sequence starting from $\mathcal{U}(S'_i)$, such that both sequences result in a common state S_s. The proof is by case analysis on evaluation derivations $S_i \Rightarrow_{is} S'_i$. In most cases, this property is straightforward because most the rules in the safe dynamic and instrumented semantics are analogous. Thus for analogous rule $S_i \Rightarrow_{is} S'_i$, we show that applying the analogous rule in the safe dynamic semantics to the transform of S_i results in the transform of S'_i (i.e. $\mathcal{U}(S_i) \Rightarrow^1_{sd} \mathcal{U}(S'_i)$). The following rules are not analogous: $unblock$, $grantmain$, $grant$, try.

The $unblock$ rule in the instrumented semantics unblocks a computation before its future completes as long as the computation's future is guaranteed not to raise an escaping exception or exit the program. Our proof leverages this guarantee. While the $unblock$ rule of the safe dynamic semantics may not apply to $\mathcal{U}(S_i)$, evaluating the future computation under a sequence of $\Rightarrow_{sd}$ rules to a value will result in a state S_s, such that the $unblock$ rule to applies to S_s. Rules $grantmain$ and $grant$ are trivial because under both rules if $S_i \Rightarrow_{is} S'_i$, then $\mathcal{U}(S_i) = \mathcal{U}(S'_i)$. Proving the try rule is similar to the $unblock$ rule. Intuitively, the proof demonstrates that runtime term $(\texttt{try let } x = \texttt{future}(N) \texttt{ in } N' \texttt{ handle } X \mapsto \ldots)$ and runtime term $(\texttt{let } x = \texttt{future}(N) \texttt{ in try } N' \texttt{ handle } X \mapsto \ldots)$ are equivalent as long as N and any future spawned from N do not raise escaping exception X. The instrumented semantics allows hoisting a future from a `try` block's evaluation context only when the static instrumentation and runtime determine it will not require the handler.

Theorem 2. *If $F_{is}(M) = R$, then $F_{sd}(M) = R$.*
Evaluating program M under the instrumented semantics will have the same result as evaluating M under the safe dynamic semantics. The proof is by induction on the length of $\Rightarrow_{is}$ evaluation sequences. The base case is demonstrated by Lemma 3 and 4. Instantiating the inductive hypothesis and Lemma 4 proves the inductive case.

5 Related Work and Conclusions

Futures were first introduced in Multilisp [3] as a high level concurrency abstraction for functional languages. Implementation of futures has been well-studied in the context of functional languages [4, 5] and future-like concurrency constructs have emerged in many multithreaded languages. Recent proposals [13, 14] that have future-like constructs do not guarantee safety of the kind provided by our solution.

In [10], deterministic execution of Java programs equipped with futures is enforced using a dynamic analysis that tracks accesses and updates by futures and their continuations; while this techniques deals with side-effects to shared fields, it does not enforce equivalence between a sequential and future-annotated Java program in the presence of exceptions. In [9], a static analysis and program transformation to provide coordination between futures and their continuations is given. [11] is closest in spirit to our work; their implementation is similar to the safe dynamic semantics presented here, but significantly less precise than the instrumented semantics.

The formal semantics of futures have been studied in [1, 2]. Their work develops a semantic framework for an idealized language with futures, but the results do not consider how to enforce safety (i.e. determinism) in the presence of exceptions. More recently, a formal semantics for an object-oriented language with active objects, asynchronous method calls and futures was presented in [12], but this presentation does not consider enforcing determinism or deal with exceptions.

This paper presents a formulation of safe futures for a higher-order language with first-class exceptions, via a combination of a static analysis to instrument programs with information about when exceptions may or may not be raised, and an operational semantics that leverages this instrumentation to extract concurrency without violating safety. We believe our results provide a precise basis for implementations of safe futures in realistic languages that support expressive control-flow abstractions.

References

1. C. Flanagan and M. Felleisen. The semantics of future and its use in program optimizations. In *Conference Record of POPL '95*, pages 209–220, San Francisco, California, 1995.
2. C. Flanagan and M. Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, 1999.
3. R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM TOPLAS*, 7(4):501–538, October 1985.
4. D. Kranz, R. H. Halstead Jr., and E. Mohr. Mul-T: A high-performance parallel Lisp. In *PLDI '89*, volume 24, pages 81–90, July 1989.
5. R. Mohr, D. Kranz, and R. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *ACM LFP Programming*, pages 185–197, June 1990.
6. O. Shivers. Control-Flow Analysis of Higher-Order Languages or Taming Lambda. PhD thesis, Carnegie Mellon University, May 1991.
7. J. Palsberg. Closure analysis in constraint form. In *ACM TOPLAS*, 47–62, 1995.
8. A. Navabi and S. Jagannathan. Exceptionally Safe Futures. Tech. Report CSD TR #08-027, Purdue University Department of Computer Science.
9. A. Navabi, X. Zhang, and S. Jagannathan. Quasi-static scheduling for safe futures. In *PPOPP '08:*, pages 23–32, 2008.
10. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for java. In *OOPSLA '05:*, pages 439–453. ACM Press, 2005.
11. L. Zhang, C. Krintz, and P. Nagpurkar. Supporting Exception Handling for Futures in Java In *PPPJ*, pages 175–184, 2007.
12. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future In *ESOP '07:*, pages 316–330, 2007.
13. E. Allan, D. Chase, V. Luchangco, J. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. The Fortress Language Specification Version 1.0. Tech. report, Sun Microsystems, 2008.
14. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538, 2005.