Type-Directed Continuation Allocation*

Zhong Shao and Valery Trifonov

Dept. of Computer Science
Yale University
New Haven, CT 06520-8285
{shao,trifonov}@cs.yale.edu

Abstract. Suppose we translate two different source languages, L_1 and L_2, into the same intermediate language; can they safely interoperate in the same address space and under the same runtime system? If L_1 supports first-class continuations (call/cc) and L_2 does not, can L_2 programs call arbitrary L_1 functions? Would the fact of possibly calling L_1 impose restrictions on the implementation strategy of L_2? Can we compile L_1 functions that do not invoke call/cc using more efficient techniques borrowed from the L_2 implementation? Our view is that the implementation of a common intermediate language ought to support the so-called *pay-as-you-go efficiency*: first-order monomorphic functions should be compiled as efficiently as in C and assembly languages, even though they may be passed to arbitrary polymorphic functions that support advanced control primitives (e.g. call/cc). In this paper, we present a typed intermediate language with effect and resource annotations, ensuring the safety of inter-language calls while allowing the compiler to choose continuation allocation strategies.

1 Introduction

Safe interoperability requires resolving a host of issues including mixed data representations, multiple function calling conventions, and different implementation protocols. Existing approaches to language interoperability either separate code written in different languages into different address spaces or have the unsafe, ad hoc and insecure foreign function call interface.

We position our further discussion of language interoperability in the context of a system hosting multiple languages, each safe in isolation. The supported languages may range from first-order monomorphic (*e.g.* a safe subset of C, or safe-C for short) to higher-order languages with advanced control, *e.g.* ML with first-class continuations. We assume that all languages have type systems which ensure runtime safety of accepted programs. In other words, in this paper we do not attempt to solve the problem of cooperating safely with programs written in unsafe languages, which in general can

* This research was sponsored in part by the DARPA ITO under the title "Software Evolution using HOT Language Technology", DARPA Order No. D888, issued under Contract No. F30602-96-2-0232, and in part by an NSF CAREER Award CCR-9501624, and NSF Grant CCR-9633390. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

only be achieved at the expense of "sandboxing" the unsafe calls or complex and incomplete analyses of the unsafe code.

We believe that interoperability requires a serious and more formal treatment. As a first step, this paper describes a novel type-based technique to support principled language interoperation among languages with different protocols for allocation of activation records. Our framework allows programs written in multiple languages with overlapping features to interact with each other safely and reliably, yet without restricting the expressiveness of each language.

An interoperability scheme for activation record allocation should be

- safe: it should not be possible to violate the runtime safety of a language by calling a foreign function;
- expressive: the scheme should allow inter-language function calls;
- efficient: a language implementation should not be forced to use suboptimal methods for its own features in order to provide support for other languages' features. For instance a language that does not use call/cc should not have to be implemented using heap-based allocation of activation records.

Our solution is to ensure safety by using a common typed intermediate language [21] into which all of the source languages are translated. To maintain safety in an expressive interoperability scheme the type system is extended with annotations of the *effects* of the evaluation of a term, e.g. an invocation of call/cc, and polymorphic types with effect variables, allowing a higher-order function to be invoked with arguments coming from languages with different sets of effects. The central novelty of our approach is the introduction of annotations of the *resources* necessary for the realization of the effects of an evaluation; for instance a continuation heap may be required when invoking call/cc. Thus our type system can be used to support implementation efficiency by keeping track of the available language-dependent resources, and safety by allowing semantically correct inter-language function calls but banning semantically incorrect ones. In addition to providing safety, making resource handling explicit also opens new opportunities for code optimization beyond what a foreign function call mechanism can offer.

A common intermediate language like FLINT [20, 21] will likely support a very rich set of features to accommodate multiple source languages. Some of these features may impose implementation restrictions; for example, a practical implementation of first-class continuations (as in SML/NJ or Scheme) often requires the use of advanced stack representations [8] or heap-based activation records [22]. However in some cases stack-based allocation may be more efficient, and ideally we would like to have a compiler that can take advantage of it as long as this does not interfere with the semantic correctness of first-class continuations. Similarly, when compiling a simple safe-C-like language with no advanced control primitives (e.g., call/cc) into FLINT, we may prefer to compile it to code that uses the simple sequential stack of standard C; programs written in ML or Scheme using these safe-C functions must then follow the same allocation strategy when invoking them. This corresponds to the typical case of writing low-level systems modules in C and providing for their use in other languages, therefore we assume this model in the sequel, but the dual problem of compiling safe-C functions

calling arbitrary ML functions by selectively imposing heap allocation on safe-C is similarly represented and solved within our system.

Thus our goal is efficient and expressive interoperability between code fragments written in languages using possibly different allocation disciplines for activation records, for instance, ML with heap allocation and safe-C with stack allocation. The following properties of the interoperability framework are essential for achieving this goal:

- ML and safe-C code should interoperate safely with each other within the same address space.
- All invocations of safe-C functions in ML functions should be allowed (provided they are otherwise type-correct).
- Only the invocations of ML functions that do not capture continuations should be allowed in safe-C functions.
- Any activation record that can potentially be captured as part of a first-class continuation should always be allocated on the heap (or using some fancy stack-chunk-based representations [8]).
- It should be possible to use stack allocation for activation records of ML functions when they are guaranteed not to be captured with a first-class continuation.
- The selection of allocation strategy should be decoupled from the actual function call.

The last property gives the compiler the freedom to switch allocation strategies more efficiently, instead of following a fixed foreign function interface mechanism. For example, an implementation of ML may use heap allocation of activation records by default to provide support for continuation capture. However, in cases when the compiler can prove that a function's activation record is not going to be accessible from any captured continuation, its allocation discipline is ambiguous; stack allocation may be preferred if the function invokes, or is invoked by, safe-C functions which use stack allocation. This specialization of code to a different allocation strategy effectively creates regions of ML code compiled in "safe-C mode" with the aim of avoiding the switch between heap and stack allocation on every cross-language call. In general, the separation of the selection of allocation strategy from the call allows its treatment as a commodity primitive operation and subjects it to other code-motion optimizations, e.g. hoisting it out of loops.

The proposed method can be applied to achieving more efficient interoperability with existing foreign code as well, although obviously in this case the usual friction between safety and efficiency can only be eased but not removed. In particular the possibility to select the allocation strategy switch point remains, thus higher efficiency can still be achieved while satisfying a given safety policy by specializing safe code to "unsafe mode" (e.g. for running with stack allocation within a sand-box).

2 A Resourceful Intermediate Language

To satisfy the requirements for efficient interoperability, outlined in the previous section, we define an A-normal-form-based typed intermediate language *RL* (Figure 1) with types having effect and resource annotations. Intuitively, an effect annotation such

as CC indicates that a computation may capture a continuation by performing call/cc; a resource annotation such as H (continuation heap) or S (continuation stack) means that the corresponding runtime resource must be available to the computation.[1] Non-trivial effects can be primitive, effect variables, or unions of effects; commutativity and associativity of the union with $\emptyset$ as a unit are consistent with the typing rules and we assume them for brevity of notation. Each effect can only occur when the proper resources are available, e.g. CC would require the use of heap-based activation record allocation. Both the effect and resource usage annotations are inferred during the translation from the source language to the intermediate language, and can be used to assist code generation and to check the validity of cross-language function calls.

RESOURCES			
	$r ::=$	S	stack continuation allocation
		$\mid \mathsf{H}$	heap continuation allocation
EFFECTS			
	$\mu ::=$	$\emptyset$	none
		$\mid \mathsf{CC}$	call with current continuation
		$\mid t$	effect variable, $t \in EffVar$
		$\mid \mu \vee \mu$	union of effects
TYPES			
	$Typ \ni \sigma ::=$	β	where $\beta \in BasicTyp$
		$\mid \sigma \xrightarrow[\mu]{r} \sigma$	resource/effect-annotated function type
		$\mid \sigma\ {}^{r}\mathsf{cont}$	resource-annotated continuation type
		$\mid \forall t \leq r. \sigma$	bounded effect-polymorphic type
VALUES AND TERMS			
	$v ::=$	c	constant $c \in Const$
		$\mid x$	variable $x \in Var$
		$\mid \lambda^r x : \sigma. e$	resource-annotated abstraction
		$\mid \Lambda t \leq r. v$	bounded effect abstraction
		$\mid x[\mu]$	effect application
	$e ::=$	$\mathbf{let}^r x = e\ \mathbf{in}\ e$	resource-annotated binding
		$\mid \langle v \rangle^r$	resource-annotated value
		$\mid \langle e \rangle_\mu$	adding spurious effects
		$\mid \mathbf{use}^r (e)$	resource selection
		$\mid @\ x\ x$	application
		$\mid \mathbf{callcc}\ x \mid \mathbf{throw}[\sigma]\ x\ x$	first class continuations

Fig. 1. Syntax of a resource-aware intermediate language *RL*

The resources required and effects produced by a function are made explicit in its type. A continuation can potentially produce all effects possible with the set of resources available at the point of its capture; for that reason continuation types only have a resource annotation.

[1] In this paper, we focus on application of this system to interoperability issues related to continuation allocation, but more diverse sets of resources will be necessary in a realistic language.

Function abstractions are annotated with the resources they may require and will maintain. In a higher-order language the effect of the evaluation of a function application may depend on the effects of its functional arguments; this dependence is expressed by means of effect polymorphism. Polymorphic abstractions introduce variables ranging over the set of possible effects of the term. Since the possible effects are determined by the available resources, we have *bounded effect polymorphism*; the relation $\mu \leq r$ (defined in the context of an effect environment in Figure 3) reflects the dependence between effects and resources, e.g. that **callcc** can only be performed if continuations are heap-allocated. The effect application $x[\mu]$ instantiates the body of the polymorphic abstraction to which x is bound. The language construct $\mathbf{use}^r\ (e)$ serves to mark the point where a change in the allocation strategy for activation records is required. Instead of having effect subsumption the language is equipped with a construct $\langle e \rangle_\mu$ for explicitly increasing the set of effects of e to include μ.

Example 1. The use of resource annotations to select allocation strategies is shown in the *RL* code below which includes extra type annotations for clarity.

$$
\begin{array}{l}
\mathbf{let}^{\mathsf{H}} \\
\quad \mathsf{applyToInt} \quad = \langle \Lambda \mathsf{t} \leq \mathsf{H}.\ \lambda^{\mathsf{H}}\, \mathsf{f}{:}\mathsf{Int} \xrightarrow[\mathsf{t}]{\mathsf{H}} \mathsf{Int}.\ @\ \mathsf{f}\ 42 \rangle^{\mathsf{H}} \\
\qquad\qquad\qquad\quad : \forall \mathsf{t} \leq \mathsf{H}.\ (\mathsf{Int} \xrightarrow[\mathsf{t}]{\mathsf{H}} \mathsf{Int}) \xrightarrow[\mathsf{t}]{\mathsf{H}} \mathsf{Int} \\
\quad \mathsf{add1\_CC} \quad = \langle \lambda^{\mathsf{H}}\, \mathsf{x}{:}\mathsf{Int}. \\
\qquad\qquad\qquad\quad \mathbf{let}^{\mathsf{H}}\ \mathsf{c} = \langle \lambda^{\mathsf{H}}\, \mathsf{k}{:}\ \mathsf{Int}\ {}^{\mathsf{H}}\mathsf{cont}. \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}^{\mathsf{H}}\ \mathsf{z} = @\ \mathsf{succ}\ \mathsf{x}\ \mathbf{in}\ \mathbf{throw}[\mathsf{Int}]\ \mathsf{k}\ \mathsf{z} \rangle^{\mathsf{H}} \\
\qquad\qquad\qquad\quad \mathbf{in}\ \mathbf{callcc}\ \mathsf{c} \rangle^{\mathsf{H}} \\
\qquad\qquad\qquad\quad : \mathsf{Int} \xrightarrow[\mathsf{CC}]{\mathsf{H}} \mathsf{Int} \\
\quad \mathsf{add1\_Pure} \quad = \langle \lambda^{\mathsf{S}}\, \mathsf{x}{:}\mathsf{Int}.\ @\ \mathsf{succ}\ \mathsf{x} \rangle^{\mathsf{H}} \\
\qquad\qquad\qquad\quad : \mathsf{Int} \xrightarrow[\emptyset]{\mathsf{S}} \mathsf{Int} \\
\quad \mathsf{add1\_Wrapped} = \langle \lambda^{\mathsf{H}}\, \mathsf{x}{:}\mathsf{Int}.\ \mathbf{use}^{\mathsf{S}}\ (@\ \mathsf{add1\_Pure}\ \mathsf{x}) \rangle^{\mathsf{H}} \\
\qquad\qquad\qquad\quad : \mathsf{Int} \xrightarrow[\emptyset]{\mathsf{H}} \mathsf{Int} \\
\mathbf{in}\ @\ (\mathsf{applyToInt}[\mathsf{CC}])\ \mathsf{add1\_CC}\,; \\
\quad @\ (\mathsf{applyToInt}[\emptyset])\ \mathsf{add1\_Wrapped}
\end{array}
$$

The function applyToInt is polymorphic in the effect of its parameter, but the parameter's resource requirements are fixed – it must use heap allocation. We consider two applications of applyToInt. The argument in the first, add1_CC, is a function invoking **callcc**, which consequently uses heap allocation; on the other hand the argument in the second application, add1_Pure, is pure and uses stack allocation. It is therefore incorrect to apply applyToInt to add1_Pure. We use a wrapper to coerce it to the proper type:

we apply applyToInt to add1_Wrapped whose activation record is heap-allocated, and whose function is to switch to stack allocation (via $\mathbf{use}^S$) before calling add1_Pure. Heap allocation is resumed upon return from add1_Pure.

3 Two Source Languages

To further illustrate the advantages of this system we consider the problem of translating into *RL* two source languages (Figure 2): a language *HL* with control operators (**callcc** and **throw**), implemented using heap-based allocation of activation records, and a language *SL* which always uses stack allocation. *HL* also allows declaring at the top of a program the identifiers of entities imported from *SL* code. The type systems of these languages are assumed monomorphic for simplicity, since polymorphism in types is largely orthogonal to the effect polymorphism of *RL*.

SL TYPES	$\tau_{SL} ::= \beta \mid \tau_{SL} \to \tau_{SL}$
SL TERMS	$e_{SL} ::= c \mid x \mid \lambda x : \tau_{SL}.\, e_{SL} \mid e_{SL}\, e_{SL} \mid \mathbf{let}\ x = e_{SL}\ \mathbf{in}\ e_{SL}$
HL TYPES	$\tau_{HL} ::= \beta \mid \tau_{HL} \to \tau_{HL} \mid \tau_{HL}\ \mathsf{cont}$
HL TERMS	$e_{HL} ::= c \mid x \mid \lambda x : \tau_{HL}.\, e_{HL} \mid e_{HL}\, e_{HL} \mid \mathbf{let}\ x = e_{HL}\ \mathbf{in}\ e_{HL}$
	$\mid \mathbf{callcc}\ e_{HL} \mid \mathbf{throw}[\tau_{HL}]\, e_{HL}\, e_{HL}$
HL PROGRAMS	$p_{HL} ::= e_{HL} \mid \mathbf{external}(SL)\ x : \tau_{SL}\ \mathbf{in}\ p_{HL}$

Fig. 2. Syntax of the source languages *SL* and *HL*

The resource annotations in *RL* provide information about handling of the stack and heap resources, necessary in the following situations:

- when calling from *HL* a function written in *SL*, which may require switching from heap allocation of activation records to allocation on the stack used by *SL*; the heap resource must be preserved for use upon return from *SL* code.
- when calling an *HL* function from *SL* code, which is only semantically sound when the evaluation of the function does not capture a continuation, since part of the continuation data is stack-allocated; the type system maintains information about the possible effects of the evaluation, in this case whether **callcc** might be invoked.
- when selecting an allocation strategy for *HL* functions called (directly or indirectly) from within *SL* code; either their activation records must be allocated on the *SL* stack, or the latter must be preserved and restored upon return to *SL*.
- when selecting an allocation strategy for *HL* code invoking *SL* functions but not **callcc**, in order to optimize resource handling.

Example 2. Consider a program consisting of a main fragment in *HL* invoking the **external** *SL* function applyToInt with the *HL* function add1 as an argument; the call is meaningful because add1 does not invoke **callcc**. Only the *SL* type of the external function is given to the *HL* program which is separately compiled without access to the detailed effect annotations inferred from the code of the *SL* fragment.

SL fragment applyToInt:

$\lambda f : \mathsf{Int} \to \mathsf{Int}.\ \mathsf{succ}\ (f\ 42)$

The result of its separate compilation into *RL*, which uses stack allocation (for details of the translation we refer the reader to Section 5) is

$$\mathsf{applyToInt} = \Lambda t \leq S.\ \lambda^{S} f{:}\mathsf{Int} \xrightarrow[t]{S} \mathsf{Int}.\ \mathbf{let}^{S}\ x = @\ f\ 42\ \mathbf{in}\ @\ \mathsf{succ}\ x$$

$$: \forall t \leq S.\ (\mathsf{Int} \xrightarrow[t]{S} \mathsf{Int}) \xrightarrow[t]{S} \mathsf{Int}$$

HL fragment main:

$\mathbf{external}(SL)\ \mathsf{applyToInt} :\ (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$
$\mathbf{in}\ \mathbf{let}\ \mathsf{add1} = \lambda x : \mathsf{Int}.\ \mathsf{succ}\ x$
$\mathbf{in}\ \mathsf{applyToInt}\ \mathsf{add1}$

The result of its separate compilation into *RL* is

$$\mathsf{main} = \lambda^{H}\ \mathsf{applyToInt}{:}\forall t \leq S.\ (\mathsf{Int} \xrightarrow[t]{S} \mathsf{Int}) \xrightarrow[\emptyset]{S} \mathsf{Int}.$$

$$\mathbf{let}^{H}$$

$$\mathsf{applyToInt\_H} = \langle \Lambda t \leq S.$$

$$\lambda^{H}\ f{:}\mathsf{Int} \xrightarrow[t]{H} \mathsf{Int}.$$

$$\mathbf{let}^{H}\ f\_S = \langle \lambda^{S}\ x{:}\mathsf{Int}.\ \mathbf{use}^{H}\ (@\ f\ x) \rangle^{H}$$

$$\mathbf{in}\ \mathbf{use}^{S}\ (@\ (\mathsf{applyToInt}[t])\ f\_S) \rangle^{H}$$

$$: \forall t \leq S.\ (\mathsf{Int} \xrightarrow[t]{H} \mathsf{Int}) \xrightarrow[\emptyset]{H} \mathsf{Int}$$

$$\mathsf{add1} = \langle \lambda^{H}\ x{:}\mathsf{Int}.\ @\ \mathsf{succ}\ x \rangle^{H}$$

$$: \mathsf{Int} \xrightarrow[\emptyset]{H} \mathsf{Int}$$

$$\mathbf{in}\ @\ \mathsf{applyToInt\_H}[\emptyset]\ \mathsf{add1}$$

$$: \left(\forall t \leq S.\ (\mathsf{Int} \xrightarrow[t]{S} \mathsf{Int}) \xrightarrow[\emptyset]{S} \mathsf{Int} \right) \xrightarrow[\emptyset]{H} \mathsf{Int}$$

The translation infers polymorphic effect types using a simplified version[2] of standard effect inference [23]. The resource annotations are fixed by the source language; the type of an external *SL* function in an *HL* program is annotated with the *SL* resources. In the code produced after translation the external functions are coerced to match the resources of *HL* using automatically generated wrappers. In the above code, the parameter f of applyToInt_H is wrapped to f_S before passing it to applyToInt; the function of the wrapper is to switch from the stack allocation discipline used by *SL* to heap allocation before invoking the code for f, and resume stack allocation upon return. Dually, the call to applyToInt itself is wrapped to enable stack allocation inside *HL* code.

[2] As presented here our system does not keep track of regions associated with effects.

Since the full *RL* type of the *SL* fragment is not available to it, the effect inference must conseratively approximate the effects of the *SL* functions. It treats the external applyToInt in the *HL* fragment as an effect-polymorphic parameter in order to allow its invocations with arguments with different effects. The price we pay for inference with this polymorphism in the case of separate compilation is that we assume that the effects of these invocations are the maximal allowed with the resources shared between the languages (in Example 2 we lose no precision since *SL* has no effects, but the approximation is reflected in the effect annotation $\emptyset$ of the type of the parameter of main). The following code, constructed mechanically given the inferred and expected types of applyToInt, coerces the actual type of applyToInt to the approximation used in the typing of main and performs the top-level application, thus linking the modules.

$$\begin{array}{l}\textbf{let}^{\mathsf{H}}\\ \quad \mathsf{applyToInt\_Glue} = \langle \Lambda \mathsf{t} \leq \mathsf{S}.\ \lambda^{\mathsf{S}}\, \mathsf{f}{:}\mathsf{Int} \xrightarrow[\mathsf{t}]{\mathsf{S}} \mathsf{Int}.\ \langle @\ \mathsf{applyToInt}[\mathsf{t}]\ \mathsf{f}\rangle_{\emptyset}\rangle^{\mathsf{H}}\\ \qquad\qquad : \forall \mathsf{t} \leq \mathsf{S}.\ (\mathsf{Int} \xrightarrow[\mathsf{t}]{\mathsf{S}} \mathsf{Int}) \xrightarrow[\emptyset]{\mathsf{S}} \mathsf{Int}\\ \textbf{in}\ @\ \mathsf{main}\ \mathsf{applyToInt\_Glue}\end{array}$$

More precise inference of the resulting effects is possible when the external function is a pre-compiled library routine whose *RL* type (with its precise effect annotations) is available when compiling main. In those cases we can take advantage of the let-polymorphism in inferring a type of main (in a setting similar to that of Example 1). However even the approximated effects obtained during separate compilation carry information that can be exploited for the optimization of inter-language calls, observing that the range of effects of a function is limited by the resources of its source language. In Example 2, after inlining and applying results of Section 4.4 (Theorem 2), the code for main can be optimized to eliminate the unnecessary switch to heap allocation in the instance of f_S. This yields

$$\begin{array}{l}\mathsf{main} = \langle \lambda^{\mathsf{H}}\, \mathsf{applyToInt}{:}\forall \mathsf{t} \leq \mathsf{S}.\ (\mathsf{Int} \xrightarrow[\mathsf{t}]{\mathsf{S}} \mathsf{Int}) \xrightarrow[\emptyset]{\mathsf{S}} \mathsf{Int}.\\ \qquad \textbf{let}^{\mathsf{H}}\\ \qquad\quad \mathsf{add1} \quad = \langle \lambda^{\mathsf{H}}\, \mathsf{x}{:}\mathsf{Int}.\ @\ \mathsf{succ}\ \mathsf{x}\rangle^{\mathsf{H}} \qquad \text{(* now dead code *)}\\ \qquad\quad \mathsf{add1\_S} = \langle \lambda^{\mathsf{S}}\, \mathsf{x}{:}\mathsf{Int}.\ @\ \mathsf{succ}\ \mathsf{x}\rangle^{\mathsf{H}}\\ \qquad \textbf{in use}^{\mathsf{S}}\ (@\ (\mathsf{applyToInt}[\emptyset])\ \mathsf{add1\_S})\rangle^{\mathsf{H}}\end{array}$$

Thus the *HL* function add1 has been effectively specialized for the stack allocation strategy used by *SL*.

Example 3. Another optimization is merging of regions with the same resource requirements, illustrated on the following *HL* code fragment.

$$\textbf{external}(\mathit{SL})\ \mathsf{intFn}\ :\ \mathsf{Int} \to \mathsf{Int}\ \textbf{in}\ \mathsf{intFn}\ (\mathsf{intFn}\ 42)$$

which is naively translated to the *RL* function (shown after inlining of the parameter wrapper)

$$\Lambda t \leq \mathsf{S}.\ \lambda^{\mathsf{H}}\,\mathsf{intFn} : \mathsf{Int} \xrightarrow[t]{\mathsf{S}} \mathsf{Int}.$$
$$\quad \mathbf{let}^{\mathsf{H}}\ \mathsf{x} = \langle \mathbf{use}^{\mathsf{S}}\ (@\,\mathsf{intFn}\ 42)\rangle^{\mathsf{H}}$$
$$\quad \mathbf{in}\ \mathbf{use}^{\mathsf{S}}\ (@\,\mathsf{intFn}\ \mathsf{x})$$

After combining the two $\mathbf{use}^{\mathsf{S}}\ (\cdot)$ constructs the equivalent *RL* term is

$$\Lambda t \leq \mathsf{S}.\ \lambda^{\mathsf{H}}\,\mathsf{intFn} : \mathsf{Int} \xrightarrow[t]{\mathsf{S}} \mathsf{Int}.$$
$$\quad \mathbf{use}^{\mathsf{S}}\ (\ \mathbf{let}^{\mathsf{S}}\ \mathsf{x} = \langle @\,\mathsf{intFn}\ 42\rangle^{\mathsf{S}}\ \mathbf{in}\ @\,\mathsf{intFn}\ \mathsf{x})$$

A generalization of this transformation makes possible lifting of $\mathbf{use}^{r}\ (\cdot)$ constructs out of a loop when the resources r are sufficient for all effects of the loop. Since in general a resource wrapper must restore resources upon return, a tail call moved into its scope effectively becomes non-tail; thus lifting a wrapper's scope over a recursive tail call is only useful when the wrapper is lifted out of the enclosing function as well, i.e. out of the loop.

4 Semantics of *RL*

4.1 Static Semantics

Correctness of resource use is ensured by the type system shown in Figure 3, which keeps track of the resources necessary for the evaluation of a term and a conservative estimate of the effects of the evaluation.

An effect environment Δ specifies the resource bounds of effect variables introduced by effect abstractions and effect-polymorphic types. The rules for effect sequents reflect the dependence of effects on resources (in this language this boils down to the dependence of the call/cc effect CC on the heap allocation resource H) and form the basis of effect polymorphism. The function MaxEff yields the maximal effect possible with a given resource; in this system we have $\mathit{MaxEff}(\mathsf{S}) = \emptyset$ and $\mathit{MaxEff}(\mathsf{H}) = \mathsf{CC}$. Rule (Eff-max) effectively states that the resource r' can be used instead of resource r if r' provides for all effects possible under r.

In the sequents assigning types to values and terms the type environment Γ maps free variables to types. Type judgments for values associate with a value v and a pair of environments Δ and Γ only a type σ, since values have no effects and therefore their evaluation requires no resources of the kind we control. The function θ maps constants to their predefined types.

Sequents for terms have the form $r; \Delta; \Gamma \vdash^{e} e : \frac{}{\mu}\sigma$, where r represents the available allocation resource, σ is the type of e, and μ represents the effects of its evaluation. Rules (Exp-let) and (Exp-val) establish the correspondence between the resource annotations in these constructs and the currently available allocation resource; the effect of lifting a value to a term is none, while the effect of sequencing two computations via **let** is the union of their effects. Any effect allowed with the current resource may be added to the effects of a term using rule (Exp-spurious).

The central novelty is the $\mathbf{use}^{r'}\ (\cdot)$ construct for resource manipulation; its typing rule (Exp-use) imposes the crucial restriction that the effect μ of the term e must be

EFFECT ENVIRONMENT FORMATION

(**Env-eff-empty**)
$$\vdash^{\Delta} \emptyset$$

(**Env-eff-ext**)
$$\frac{\vdash^{\Delta} \Delta}{\vdash^{\Delta} \Delta_t,\ t \leq r}$$

TYPE ENVIRONMENT FORMATION

(**Env-typ-empty**)
$$\frac{\vdash^{\Delta} \Delta}{\Delta \vdash^{\Gamma} \emptyset}$$

(**Env-typ-ext**)
$$\frac{\Delta \vdash^{\Gamma} \Gamma \quad \Delta \vdash^{\sigma} \sigma}{\Delta \vdash^{\Gamma} \Gamma_x,\ x : \sigma}$$

EFFECTS

(**Eff-empty**)
$$\frac{\vdash^{\Delta} \Delta}{\Delta \vdash^{\mu} \emptyset \leq r}$$

(**Eff-CC**)
$$\frac{\vdash^{\Delta} \Delta}{\Delta \vdash^{\mu} \mathsf{CC} \leq \mathsf{H}}$$

(**Eff-var**)
$$\frac{\vdash^{\Delta} \Delta \quad \Delta(t) = r}{\Delta \vdash^{\mu} t \leq r}$$

(**Eff-combine**)
$$\frac{\vdash^{\mu} \mu' \leq r,\ \mu'' \leq r}{\Delta \vdash^{\mu} \mu' \vee \mu'' \leq r}$$

(**Eff-max**)
$$\frac{\Delta \vdash^{\mu} \mu \leq r \quad \Delta \vdash^{\mu} \mathit{MaxEff}(r) \leq r'}{\Delta \vdash^{\mu} \mu \leq r'}$$

VALUES

(**Val-const**)
$$\frac{\Delta \vdash^{\Gamma} \Gamma}{\Delta; \Gamma \vdash^{v} c : \theta(c)}$$

(**Val-var**)
$$\frac{\Delta \vdash^{\Gamma} \Gamma \quad \Gamma(x) = \sigma}{\Delta; \Gamma \vdash^{v} x : \sigma}$$

(**Val-abs**)
$$\frac{\Delta \vdash^{\Gamma} \Gamma \quad \Delta \vdash^{\sigma} \sigma \quad r; \Delta; \Gamma_x,\ x : \sigma \vdash^{e} e : \underset{\mu}{-}\sigma'}{\Delta; \Gamma_x \vdash^{v} \lambda^{r}\, x{:}\sigma.\ e : \sigma \xrightarrow[\mu]{r} \sigma'}$$

(**Val-poly**)
$$\frac{\Delta \vdash^{\Gamma} \Gamma \quad \Delta_t,\ t \leq r; \Gamma \vdash^{v} v : \sigma}{\Delta; \Gamma \vdash^{v} \Lambda t \leq r.\ v : \forall t \leq r.\ \sigma}$$

(**Val-tapp**)
$$\frac{\Gamma(x) = \forall t \leq r.\ \sigma \quad \Delta \vdash^{\mu} \mu \leq r}{\Delta; \Gamma \vdash^{v} x[\mu] : [\mu/t]\sigma}$$

TYPES

(**Typ-basic**)
$$\frac{\vdash^{\Delta} \Delta}{\Delta \vdash^{\sigma} \beta}$$

(**Typ-fun**)
$$\frac{\Delta \vdash^{\mu} \mu \leq r \quad \Delta \vdash^{\sigma} \sigma,\ \sigma'}{\Delta \vdash^{\sigma} \sigma \xrightarrow[\mu]{r} \sigma'}$$

(**Typ-cont**)
$$\frac{\Delta \vdash^{\sigma} \sigma \quad \emptyset \vdash^{\mu} \mathsf{CC} \leq r}{\Delta \vdash^{\sigma} \sigma\ {}^{r}\mathsf{cont}}$$

(**Typ-poly**)
$$\frac{\vdash^{\Delta} \Delta \quad \Delta_t,\ t \leq r \vdash^{\sigma} \sigma}{\Delta \vdash^{\sigma} \forall t \leq r.\ \sigma}$$

TERMS

(**Exp-let**)
$$\frac{r; \Delta; \Gamma \vdash^{e} e : \underset{\mu}{-}\sigma \quad r; \Delta; \Gamma_x,\ x : \sigma \vdash^{e} e' : \underset{\mu'}{-}\sigma'}{r; \Delta; \Gamma \vdash^{e} \mathbf{let}^{r}\ x = e\ \mathbf{in}\ e' : \underset{\mu \vee \mu'}{-}\sigma'}$$

(**Exp-val**)
$$\frac{\Delta; \Gamma \vdash^{v} v : \sigma}{r; \Delta; \Gamma \vdash^{e} \langle v \rangle^{r} : \underset{\emptyset}{-}\sigma}$$

(**Exp-spurious**)
$$\frac{r; \Delta; \Gamma \vdash^{e} e : \underset{\mu}{-}\sigma \quad \Delta \vdash^{\mu} \mu' \leq r}{r; \Delta; \Gamma \vdash^{e} \langle e \rangle_{\mu'} : \underset{\mu \vee \mu'}{-}\sigma}$$

(**Exp-use**)
$$\frac{r'; \Delta; \Gamma \vdash^{e} e : \underset{\mu}{-}\sigma \quad \Delta \vdash^{\mu} \mu \leq r}{r; \Delta; \Gamma \vdash^{e} \mathbf{use}^{r'}\ (e) : \underset{\mu}{-}\sigma}$$

(**Exp-app**)
$$\frac{\Delta \vdash^{\Gamma} \Gamma \quad \Gamma(x) = \sigma' \xrightarrow[\mu]{r} \sigma \quad \Gamma(x') = \sigma'}{r; \Delta; \Gamma \vdash^{e} @\ x\ x' : \underset{\mu}{-}\sigma}$$

(**Exp-callcc**)
$$\frac{\Delta \vdash^{\Gamma} \Gamma \quad \Gamma(x) = \sigma\ {}^{r}\mathsf{cont} \xrightarrow[\mu]{r} \sigma}{r; \Delta; \Gamma \vdash^{e} \mathbf{callcc}\ x : \underset{\mu \vee \mathsf{CC}}{-}\sigma}$$

(**Exp-throw**)
$$\frac{\Delta \vdash^{\Gamma} \Gamma \quad \Delta \vdash^{\sigma} \sigma' \quad \Gamma(x) = \sigma\ {}^{r}\mathsf{cont} \quad \Gamma(x') = \sigma}{r; \Delta; \Gamma \vdash^{e} \mathbf{throw}[\sigma']\ x\ x' : \underset{\mathit{MaxEff}(r)}{-}\sigma'}$$

Fig. 3. The *RL* type system

supported by the resource r available before the alternative resource r' is selected. This ensures the correctness of the propagation of μ outside the scope of the $\mathbf{use}^{r'}\ (\cdot)$.

The rules for application and **callcc** set the correspondence between the available resource and the resource required by the invoked function. In addition, (Exp-callcc) and (Exp-throw) specify that the continuation type is annotated with the same resource, which is needed by the context captured in the continuation and therefore must be matched when it is reactivated. The effect of evaluating a **callcc** includes CC, while the effect of a **throw** is that of the rest of the computation, which we estimate as the maximal possible with the current resource.

By induction on the structure of a typing derivation it follows that if a term has a type in a given environment, it has exactly one type, and the presence of type annotations allows its effective computation, *i.e.* there exists a function $\mathit{EffTypeOf}$ such that

$$\mathit{EffTypeOf}\ (r,\ \Delta,\ \Gamma,\ e) = \langle \mu, \sigma \rangle \text{ if and only if } r; \Delta; \Gamma \vdash^{e} e : \underset{\mu}{-}\sigma.$$

We will also use the function TypeOf with the same arguments, returning the type σ only.

4.2 Dynamic Semantics

The operational semantics of *RL* (Figure 4) is defined by means of a variant of the tail-call-safe C_aEK machine (Flanagan *et al.* [4]). The machine configuration is a tuple $\langle e,\ E,\ O,\ \rho \rangle$ where e is the current term to be evaluated, E is the environment mapping variables to machine values, O is a heap of objects (closures), and ρ is a tuple of *machine resources*. Depending on the allocation strategy used, ρ is either a continuation stack S, recording (as in the original C_aEK machine) the context of the evaluation as a sequence of activation records, or a pair of a current continuation k and a continuation heap K. In the latter form k is a continuation handle and K is a mapping from $\mathit{ContHandles}$ to activation records which offers non-sequential access. In neither case does a function application (app) perform additional allocations of activation records, so both strategies are tail-call safe.

Machine values are either small constants or pointers into other structures where larger objects are allocated. All closures are allocated on the heap (the function γ at the bottom of the figure shows the details).

The activation records created when evaluating a $\mathbf{let}^{r}$-expression may be allocated either on the continuation heap K (transition rule ($\mathsf{let}^{\mathsf{H}}$)) or on the continuation stack S (rule ($\mathsf{let}^{\mathsf{S}}$)). An activation record represents a continuation, and in our small language there are only three possibilities: the computation either halts or continues by binding a variable to a computed value or by restoring a resource. Rules ($\mathsf{val}^{\mathsf{H}}$) and ($\mathsf{val}^{\mathsf{S}}$) perform the binding, depending on the allocation mode.

The evaluation of $\mathbf{use}^{r}\ (e)$ selects the activation record allocation strategy for e, *e.g.* $\mathbf{use}^{\mathsf{S}}\ (e)$ selects stack-based allocation for e (transition rule ($\mathsf{use}^{\mathsf{S}}$)). When the current allocation resource is already r we define $\mathbf{use}^{r}\ (\cdot)$ as a no-op; if a change of resource is performed, an activation record is pushed on (the top of) the new allocation resource. Correspondingly, heap-based allocation is restored by transition rule ($\mathsf{resume}^{\mathsf{H}}$) after the evaluation of e.

SEMANTIC DOMAINS

$MachineVal \ni w$	$::=$	$\mathsf{Const}\ c \mid \mathsf{Ptr}\ h \mid \mathsf{Cont}\ k$	machine values
E	$\in$	$Var \to MachineVal$	environment
h	$\in$	$HeapLocs$	heap locations
$Object \ni o$	$::=$	$\mathsf{Closure}\ \langle x, e, E\rangle \mid \mathsf{TyAbs}\ \langle t, r, v\rangle$	closures (objects)
O	$\in$	$HeapLocs \to Object$	object heap
k	$\in$	$ContHandles$	continuation handles
$ActRcd \ni a$	$::=$	$\mathsf{Bind}\ \langle x, e, E, k\rangle \mid \mathsf{Resume}\ S \mid \mathsf{Halt}$	activation records
K	$\in$	$ContHandles \to ActRcd$	activation record heap
S	$::=$	$\mathsf{Bind}\ \langle x, e, E, S\rangle \mid \mathsf{Resume}\ \langle k, K\rangle \mid \mathsf{Halt}$	activation record stack

TRANSITION RULES

(app) $\langle @\, x_1\ x_2,\ E,\ O,\ \rho\rangle \mapsto_1 \langle e',\ E'[x' \mapsto E(x_2)],\ O,\ \rho\rangle$
where $E(x_1) = \mathsf{Ptr}\ h,\ O(h) = \mathsf{Closure}\ \langle x', e', E'\rangle$

FOR HEAP-ALLOCATED ACTIVATION RECORDS

($\mathsf{let}^{\mathsf{H}}$) $\langle \mathbf{let}^{\mathsf{H}}\ x = e_1\ \mathbf{in}\ e_2,\ E,\ H,\ \langle k,\ K\rangle\rangle \mapsto_1$
$\langle e_1,\ E,\ H,\ \langle k',\ K[k' \mapsto \mathsf{Bind}\ \langle x, e_2, E|_{FV(e_2)-x}, k\rangle]\rangle\rangle$

($\mathsf{val}^{\mathsf{H}}$) $\langle\langle v\rangle^{\mathsf{H}},\ E,\ H,\ \langle k,\ K\rangle\rangle \mapsto_1 \langle e',\ E'[x' \mapsto w],\ O',\ \langle k',\ K\rangle\rangle$
where $K(k) = \mathsf{Bind}\ \langle x', e', E', k'\rangle,\ \langle w, O'\rangle = \gamma\,(v, E, O)$

(callcc) $\langle \mathbf{callcc}\ x,\ E,\ H,\ \langle k,\ K\rangle\rangle \mapsto_1 \langle e',\ E'[x' \mapsto \mathsf{Cont}\ k],\ O,\ \langle k,\ K\rangle\rangle$
where $E(x) = \mathsf{Ptr}\ h,\ O(h) = \mathsf{Closure}\ \langle x', e', E'\rangle$

(throw) $\langle \mathbf{throw}[\sigma]\ x_1\ x_2,\ E,\ H,\ \langle k,\ K\rangle\rangle \mapsto_1 \langle e',\ E'[x' \mapsto E(x_2)],\ O,\ \langle k',\ K\rangle\rangle$
where $E(x_1) = \mathsf{Cont}\ k_1,\ K(k_1) = \mathsf{Bind}\ \langle x', e', E', k'\rangle$

($\mathsf{use}^{\mathsf{S}}$) $\langle \mathbf{use}^{\mathsf{S}}\ (e),\ E,\ H,\ \langle k,\ K\rangle\rangle \mapsto_1 \langle e,\ E,\ H,\ \langle \mathsf{Resume}\ \langle k, K\rangle\rangle\rangle$

($\mathsf{resume}^{\mathsf{S}}$) $\langle\langle v\rangle^{\mathsf{H}},\ E,\ H,\ \langle k,\ K\rangle\rangle \mapsto_1 \langle\langle v\rangle^{\mathsf{S}},\ E,\ H,\ \langle S\rangle\rangle$
where $K(k) = \mathsf{Resume}\ S$

FOR STACK-ALLOCATED ACTIVATION RECORDS

($\mathsf{let}^{\mathsf{S}}$) $\langle \mathbf{let}^{\mathsf{S}}\ x = e_1\ \mathbf{in}\ e_2,\ E,\ H,\ \langle S\rangle\rangle \mapsto_1 \langle e_1,\ E,\ H,\ \langle \mathsf{Bind}\ \langle x, e_2, E|_{FV(e_2)-x}, S\rangle\rangle\rangle$

($\mathsf{val}^{\mathsf{S}}$) $\langle\langle v\rangle^{\mathsf{S}},\ E,\ H,\ \langle \mathsf{Bind}\ \langle x', e', E', S\rangle\rangle\rangle \mapsto_1 \langle e',\ E'[x' \mapsto w],\ O',\ \langle S\rangle\rangle$
where $\langle w, H'\rangle = \gamma\,(v, E, O)$

($\mathsf{use}^{\mathsf{H}}$) $\langle \mathbf{use}^{\mathsf{H}}\ (e),\ E,\ H,\ \langle S\rangle\rangle \mapsto_1 \langle e,\ E,\ H,\ \langle k,\ [k \mapsto \mathsf{Resume}\ S]\rangle\rangle$

($\mathsf{resume}^{\mathsf{H}}$) $\langle\langle v\rangle^{\mathsf{S}},\ E,\ H,\ \langle \mathsf{Resume}\ \langle k, K\rangle\rangle\rangle \mapsto_1 \langle\langle v\rangle^{\mathsf{H}},\ E,\ H,\ \langle k,\ K\rangle\rangle$

REPRESENTATION OF VALUES

$\gamma\,(c, E, O) = \langle \mathsf{Const}\ c,\ O\rangle \quad \gamma\,(\lambda^r\ x{:}\sigma.\ e, E, O) = \langle \mathsf{Ptr}\ h,\ O[h \mapsto \mathsf{Closure}\ \langle x, e, E|_{FV(e)-x}\rangle]\rangle$

$\gamma\,(x, E, O) = \langle E(x),\ O\rangle \qquad \gamma\,(\Lambda t \le r.\ v, E, O) = \langle \mathsf{Ptr}\ h,\ O[h \mapsto \mathsf{TyAbs}\ \langle t, r, v\rangle]\rangle$
where $h \notin Dom\,(O)$

$\gamma\,(x[\mu], E, O) = \gamma\,([\mu/t]v, E, O)$ if $E(x) = \mathsf{Ptr}\ h',\ O(h') = \mathsf{TyAbs}\ \langle t, r, v\rangle$, and $\vdash^{\mu} \mu \le r$

Fig. 4. Semantics of *RL*

Another no-op is the increase of effect sets $\langle\cdot\rangle_\mu$ which only serves type-checking purposes.

4.3 Soundness of the Type System

The type system maintains the property that the effects of well-typed programs are possible with their available resources, formalized in the following statement, proved by induction on the typing derivation.

Lemma 1. *If $r; \Delta; \Gamma \vdash^e e : \frac{}{\mu}\sigma$ is a valid typing judgment, then $\Delta \vdash^\mu \mu \leq r$.*

Semantically this behavior of well-typed programs is expressed as soundness with respect to resource use, extending the standard soundness for safety of the type system, in the following theorem.

Theorem 1. *If $r; \emptyset; \emptyset \vdash^e e : \frac{}{\mu}\sigma$, then the configuration $\langle e,\ \emptyset,\ \emptyset,\ \mathsf{Halt}^{\,r}\rangle$ either diverges or evaluates to the configuration $\langle\langle v\rangle^r,\ E,\ O,\ \langle\mathsf{Halt}^{\,r}\rangle\rangle$ (for some v, E and O), where $\mathsf{Halt}^{\,\mathsf{S}} \triangleq \langle\mathsf{Halt}\,\rangle$, and $\mathsf{Halt}^{\,\mathsf{H}} \triangleq \langle k,\ K\rangle$ for some k and K such that $K(k) = \mathsf{Halt}$.*

This result is a corollary of the standard properties of progress and subject reduction of the system, the proofs of which we sketch below. To simplify the proofs, we introduce a type-annotated version of the semantics, which maintains type information embedded in the runtime representation. Thus the representation of an abstraction in the type-annotated version is

$$\gamma\,(\lambda^r\, x{:}\sigma.\ e,\ E,\ O) = \langle \mathsf{Ptr}\ h,\ O[h \mapsto \mathsf{Closure}'\ \langle r,\ x,\ \sigma,\ e,\ E|_{FV(e)-x}\rangle]\rangle$$

In addition, the runtime environment E is extended to keep the type of each value in its codomain; the value component of E is denoted by ${}^VE$ and the type component by ${}^TE$.

The following definitions are helpful in defining typability of configurations.

Definition 1. *The* bottom $bot(\rho)$ *of an allocation resource ρ is defined as follows:*

1. *if $\rho = \langle S\rangle$, then $bot(\rho) = bot(S')$, if $S = \mathsf{Bind}\ \langle x', e', E', S'\rangle$, and $bot(\rho) = S$ otherwise;*
2. *if $\rho = \langle k, K\rangle$, then $bot(\rho) = bot(\langle k', K\rangle)$, if $K(k) = \mathsf{Bind}\ \langle x', e', E', k'\rangle$, and $bot(\rho) = K(k)$ otherwise.*

Definition 2. *The* outermost continuation heap $outerCont(\rho)$ *reachable from allocation resource ρ is*

1. *K if $\rho = \langle k, K\rangle$ and $bot(\rho) = \mathsf{Halt}$;*
2. *$outerCont(\langle S\rangle)$ if $\rho = \langle k, K\rangle$ and $bot(\rho) = \mathsf{Resume}\ S$;*
3. *$\emptyset$, if $\rho = \langle S\rangle$ and $bot(\rho) = \mathsf{Halt}$;*
4. *$outerCont(\langle k, K\rangle)$ if $\rho = \langle S\rangle$ and $bot(\rho) = \mathsf{Resume}\ \langle k, K\rangle$.*

Definition 3. *A configuration closed in type environment Γ is* typable *under resource r with a result type σ and an effect μ, written $r;\ \Gamma \vdash^c \langle e,\ E,\ O,\ \rho\rangle : \frac{}{\mu}\sigma$, if for some σ', μ'*

1. $Dom(\Gamma) \cap Dom(E) = \emptyset$; *and*
2. $r; \emptyset; \Gamma, {}^{T}E \vdash^{e} e : \frac{}{\mu'}\sigma'$; *and*
3. $\Gamma \vdash^{\rho} \langle \rho, E, O \rangle \in \sigma' \overset{r}{\underset{\mu'}{\hookrightarrow}} \sigma$; *and*
4. *for each* $x \in Dom(E)$,
   (a) *if* ${}^{V}E(x) = \mathsf{Const}\ c$, *then* ${}^{T}E(x) = \theta(c)$;
   (b) *if* ${}^{V}E(x) = \mathsf{Ptr}\ h$ *and* $O(h) = \mathsf{Closure}'\ \langle r_1, x_1, \sigma_1, e_1, E_1 \rangle$, *then* $\emptyset; {}^{T}E_1 \vdash^{v} \lambda^{r_1}\ x_1 : \sigma_1.\ e_1 : {}^{T}E(x)$, *and similarly for type abstractions;*
   (c) *if* ${}^{V}E(x) = \mathsf{Cont}\ k$, *then* ${}^{T}E(x) = \sigma_1\ {}^{r_1}\mathsf{cont}$ *and*

$$\Gamma \vdash^{\rho} \langle k,\ outerCont(\rho) \rangle, E, O \in \sigma_1 \overset{r_1}{\underset{\mu_1}{\hookrightarrow}} \sigma_1'$$

   and $\mu = \mu_1 \vee \mu_1'$, *for some* σ_1' *and* μ_1',

and $\Gamma \vdash^{\rho} \langle \rho, E, O \rangle \in \sigma' \overset{r}{\underset{\mu}{\hookrightarrow}} \sigma$ *if*

1. $r = \mathsf{S}$ *and* $\rho = \langle \mathsf{Halt} \rangle$ (i.e. *an empty stack) and* $\sigma = \sigma'$ *and* $\mu = \emptyset$; *or*
2. $r = \mathsf{S}$ *and* $\rho = \langle \mathsf{Bind}\ \langle x_1, e_1, E_1, S_1 \rangle \rangle$ *and* $\mathsf{S};\ \Gamma, x_1 : \sigma' \vdash^{e} \langle e_1, E_1, O, S_1 \rangle : \frac{}{\mu}\sigma$; *or*
3. $r = \mathsf{S}$ *and* $\rho = \langle \mathsf{Resume}\ \langle k', K' \rangle \rangle$ *and* $\Gamma \vdash^{\rho} \langle \langle k', K' \rangle, E, O \rangle \in \sigma' \overset{\mathsf{H}}{\underset{\mu}{\hookrightarrow}} \sigma$,

and similarly for $r = \mathsf{H}$.

Note that the environment may contain reachable variables bound to continuations even when the current allocation resource is a stack. Type correctness of these continuations cannot be verified with the stack resource, instead we have to find the corresponding continuation heap. However in this case the type system guarantees that the only continuation heap to which there are references in the environment is the outermost continuation heap, if such exists. The reason is that although it is possible to switch to heap allocation after executing in stack allocation mode, there are no invocations of **callcc** allowed since they would introduce the CC effect, which is not possible under the stack resource (cf. typing rule (Exp-use) in Figure 3).

We can now formulate the progress and subject reduction properties.

Lemma 2 (Progress). *If* $r; \emptyset \vdash^{c} \langle e, E, O, \rho \rangle : \frac{}{\mu}\sigma$ *where* r *corresponds to* ρ (i.e. $r = \mathsf{S}$ *if* $\rho = \langle S \rangle$, $r = \mathsf{H}$ *if* $\rho = \langle k, K \rangle$), *and* $\rho \neq \mathsf{Halt}^{\,r}$, *then there exists* C *such that* $\langle e, E, O, \rho \rangle \mapsto_1 C$.

Lemma 3 (Subject reduction). *If* $C = \langle e, E, O, \rho \rangle$ *and* $r; \emptyset \vdash^{c} C : \frac{}{\mu}\sigma$ *where* r *corresponds to* ρ, *and* $C \mapsto_1 C' = \langle e', E', O', \rho' \rangle$, *then* $r'; \emptyset \vdash^{c} C' : \frac{}{\mu'}\sigma$ *where* r' *corresponds to* ρ', $\mu = \mu' \vee \mu_1'$, *and the rule for this transition is* (callcc) *only if* $\mu = \mathsf{CC} \vee \mu''$, *for some* μ_1' *and* μ''.

In brief, in the case when $e \neq \langle v \rangle^{r}$, the proofs proceed by examining the structure of the typing derivation for $r; \emptyset; \Gamma, {}^{T}E \vdash^{e} e : \frac{}{\mu'}\sigma'$; together with condition 4 of Definition 3 this yields that the values in the environment and on the heaps have the correct shape for the appropriate transition rule. In the case when e has the form $\langle v \rangle^{r}$ the proofs inspect the structure of the derivation of $\Gamma \vdash^{\rho} \langle \rho, E, O \rangle \in \sigma' \overset{r}{\underset{\mu}{\hookrightarrow}} \sigma$, which parallels the decision tree for the transition rules (val) and (resume) and the halting state.

4.4 Resource Transformations

Effect inference and type correctness with respect to resource use allow the compiler to modify the continuation allocation strategy of a program fragment and preserve its meaning. The following definitions adapt the standard notions of ordering and observational equivalence of open terms to the resource-based system.

Definition 4. *A* context C *is a term with a hole* $\bullet$*; the result of placing a term* e *in the hole of* C *is denoted by* $C[e]$ *and may result in capturing effect and lambda variables free in* e*. The hole of a context* C *is of type* $(r, \Delta, \Gamma) \Rightarrow \frac{}{\mu}\sigma$ *if* $C[e]$ *is typeable whenever* $r; \Delta; \Gamma \vdash^{e} e : \frac{}{\mu}\sigma$.

Definition 5. $\mathsf{S}; \Delta; \Gamma \vdash^{e} e \sqsubseteq e' : \frac{}{\mu}\sigma$ *if for all contexts* C *with hole of type* $(r, \Delta, \Gamma) \Rightarrow \frac{}{\mu}\sigma$*, all typed environments* E *closing* $C[e]$ *and heaps* O *closing* E*, and continuation stacks* S*, the configuration* $\langle C[e'], E, O, \langle S \rangle\rangle$ *converges if* $\langle C[e], E, O, \langle S \rangle\rangle$ *converges. Furthermore,* $\mathsf{S}; \Delta; \Gamma \vdash^{e} e \approx e' : \frac{}{\mu}\sigma$ *if* $\mathsf{S}; \Delta; \Gamma \vdash^{e} e \sqsubseteq e' : \frac{}{\mu}\sigma$ *and* $\mathsf{S}; \Delta; \Gamma \vdash^{e} e' \sqsubseteq e : \frac{}{\mu}\sigma$.

One possible optimization is the conversion of heap-allocating code to stack-based strategy provided the code does not invoke **callcc** or **throw**, as per the following theorem.

Theorem 2. *If* $\mathsf{H}; \Delta; \Gamma \vdash^{e} e : \frac{}{\emptyset}\sigma$*, then* $\mathsf{S}; \Delta; \Gamma \vdash^{e} \mathbf{use}^{\mathsf{H}}\,(e) \approx \mathit{StkCont}_{\Delta}\,(e;\, \Gamma) : \frac{}{\emptyset}\sigma$*, where* $\mathit{StkCont}$ *is the transformation defined as follows.*

$$
\begin{aligned}
\mathit{StkCont}_{\Delta}\,(\langle v \rangle^{\mathsf{H}};\, \Gamma) &= \langle v \rangle^{\mathsf{S}} \\
\mathit{StkCont}_{\Delta}\,(\langle e \rangle_{\mu};\, \Gamma) &= \langle \mathit{StkCont}_{\Delta}\,(e;\, \Gamma) \rangle_{\mu} \\
\mathit{StkCont}_{\Delta}\,(\mathbf{use}^{\mathsf{H}}\,(e);\, \Gamma) &= \mathit{StkCont}_{\Delta}\,(e;\, \Gamma) \\
\mathit{StkCont}_{\Delta}\,(\mathbf{use}^{\mathsf{S}}\,(e);\, \Gamma) &= e \\
\mathit{StkCont}_{\Delta}\,(@\, x_1\ x_2;\, \Gamma) &= \mathbf{let}^{\mathsf{S}}\ x_1' = \langle \lambda^{\mathsf{S}}\, x_2' : \Gamma(x_2).\ \mathbf{use}^{\mathsf{H}}\,(@\, x_1\ x_2') \rangle^{\mathsf{S}} \\
&\qquad \mathbf{in}\ @\, x_1'\ x_2 \\
\mathit{StkCont}_{\Delta}\,(\mathbf{let}^{\mathsf{H}}\ x = e_1\ \mathbf{in}\ e_2;\, \Gamma) &= \mathbf{let}^{\mathsf{S}}\ x = \mathit{StkCont}_{\Delta}\,(e_1;\, \Gamma) \\
&\qquad \mathbf{in}\ \mathit{StkCont}_{\Delta}\,(e_2;\, \Gamma_x, x : \mathit{TypeOf}\ (\mathsf{H}, \Delta, \Gamma, e_2))
\end{aligned}
$$

5 Translation from *HL* to *RL*

Programs in language $\mathcal{L} \in \{HL, SL\}$ are translated into *RL* by an algorithm shown in Figure 5. The algorithm infers the effect and resource annotations of a term using fairly standard techniques. It is presented in the form of an inference system for judgments of the form $\Delta; \Gamma \vdash_{\mathcal{L}} e_{HL} \Rightarrow \Delta' \vdash e : \frac{}{\mu}\sigma$, where e_{HL}, Δ, and Γ are inputs corresponding respectively to the $\mathcal{L}$ term to translate (also overloaded to *HL* top-level programs) and the inherited effect and type environments, initially empty. The outputs of the translation are e, Δ', μ, and σ, which stand for the translated term, the inferred effect environment, and the effect and type of e in environments Δ' and Γ; thus the output of the algorithm satisfies $\mathsf{H}; \Delta'; \Gamma \vdash^{e} e : \frac{}{\mu}\sigma$. The function $\mathcal{R}$ maps a language name to the resources available to a program in this language: $\mathcal{R}(HL) = \mathsf{H}$ and $\mathcal{R}(SL) = \mathsf{S}$.

(Translate-external)

$$\frac{\sigma' = \mathit{CloseAll}\,(\mathit{Max}^{\mathsf{S}}(\mathit{Annotate}^{\mathsf{S}}(\tau,\ \mathit{Dom}\,(\Delta))),\ \mathsf{S}) \quad \sigma'' = \mathit{CloseAll}\,(\mathit{Annotate}^{\mathsf{H}}(\tau,\ \mathit{Dom}\,(\Delta)),\ \mathsf{S}) \quad \Delta;\Gamma_x, x:\sigma'' \vdash_{HL} p \Rightarrow \Delta' \vdash e' : \underset{\mu}{-}\sigma}{\begin{array}{l}\Delta;\Gamma \vdash_{HL} \mathbf{external}(SL)\ x:\tau\ \mathbf{in}\ p \\ \qquad \Rightarrow \Delta' \vdash \lambda^{\mathsf{H}}\, x{:}\sigma'.\ \mathbf{let}^{\mathsf{H}}\ x = \mathit{Wrap}^{\mathsf{H}}_{\mathsf{S}}(\bullet,\ x,\ \sigma')\ \mathbf{in}\ e' : \underset{\emptyset}{-}(\sigma' \xrightarrow[\mu]{\mathsf{H}} \sigma)\end{array}}$$

where

$$\begin{array}{rl}
\mathit{Annotate}^r\,(\beta,\ V) &= \beta \\
\mathit{Annotate}^r\,(\tau\ \mathsf{cont},\ V) &= (\mathit{Annotate}^r\,(\tau,\ V))\ {}^r\mathsf{cont} \\
\mathit{Annotate}^r\,(\tau \to \tau',\ V) &= \sigma \xrightarrow[t]{r} \sigma' \text{ where } t \in \mathit{EffVar} - V, \\
&\qquad \sigma = \mathit{Annotate}^r\,(\tau,\ V \cup \{t\}), \\
&\qquad \sigma' = \mathit{Annotate}^r\,(\tau,\ V \cup \{t\} \cup \mathit{fev}(\sigma)) \\
\mathit{Wrap}^{r'}_r\,(C,\ x,\ \forall t \le r''.\ \sigma) &= \Lambda t \le r''.\ \mathit{Wrap}^{r'}_r\,(C[\mathbf{let}^{r'}\ x' = \langle x[t]\rangle^{r'}\ \mathbf{in}\ \bullet],\ x',\ \sigma) \\
\mathit{Wrap}^{r'}_r\,(C,\ x,\ \sigma_1 \xrightarrow[\mu]{r} \sigma_2) &= \lambda^{r'}\, x_1{:}\sigma_1'.\ \mathbf{let}^{r'}\ x_1' = \langle\, \mathit{Wrap}^r_{r'}\,(\bullet,\ x_1,\ \sigma_1')\rangle^{r'} \\
&\qquad \mathbf{in}\ \mathit{Wrap}^{r'}_r\,(C[\mathbf{let}^{r'}\ x_2 = @\ x\ x_1'\ \mathbf{in}\ \bullet],\ x_2,\ \sigma_2) \\
&\qquad \text{where } \sigma_1' = \mathit{ConvertType}^{r'}_r\,(\sigma_1) \\
\mathit{Wrap}^{r'}_r\,(C,\ x,\ \beta) &= C[\langle x\rangle^{r'}]
\end{array}$$

(Translate-app)

$$\frac{\Delta;\Gamma \vdash_{\mathcal{L}} e_1 \Rightarrow \Delta_1 \vdash e_1' : \underset{\mu_1}{-}\sigma_1 \quad \Delta_1;\Gamma \vdash_{\mathcal{L}} e_2 \Rightarrow \Delta_2 \vdash e_2' : \underset{\mu_2}{-}\sigma_2 \quad \Delta';S \vdash \sigma_1 \sim (\sigma_2 \xrightarrow[t]{\mathsf{H}} \alpha) \quad t \notin \mathit{fev}(\sigma_1) \cup \mathit{fev}(\sigma_2) \cup \mathit{Dom}\,(\Delta_2) \quad x_1 \notin \mathit{FV}(e_2')}{\Delta_2 \sqcap \Delta';\Gamma \vdash_{\mathcal{L}} e_1\, e_2 \Rightarrow \Delta' \vdash \mathbf{let}^{\mathsf{H}}\ x_1 = e_1'\ \mathbf{in}\ \mathbf{let}^{\mathsf{H}}\ x_2 = e_2'\ \mathbf{in}\ @\ x_1\ x_2 : \underset{\mu_1 \vee \mu_2 \vee St}{-}S\alpha}$$

where

$$\emptyset;[\sigma/\alpha] \vdash \alpha \sim \sigma \qquad \frac{\Delta_1;S_1 \vdash \sigma_1 \sim \sigma_1' \quad \Delta_2;S_2 \vdash S_1\sigma_2 \sim S_1\sigma_2' \quad S = \mathit{mgu}(S_2\mu, S_2\mu')}{\Delta_1 \sqcap \Delta_2;S \vdash \sigma_1 \xrightarrow[\mu]{r} \sigma_2 \sim \sigma_1' \xrightarrow[\mu']{r} \sigma_2'}$$

$$\frac{\Delta;S \vdash \sigma_1 \sim \sigma_2}{\Delta;S \vdash \sigma_1\ {}^r\mathsf{cont} \sim \sigma_2\ {}^r\mathsf{cont}} \qquad \frac{\Delta;S \vdash \sigma \sim \sigma' \quad \Delta' = \mathit{MinEnv}(St \le r)}{\Delta \sqcap \Delta';S\backslash_{\{t\}} \vdash \forall t \le r.\ \sigma \sim \sigma'}$$

$$\begin{array}{ll}
\mathit{MinEnv}\,(t \le r) = t \le r & \mathit{MinEnv}\,(\mu_1 \vee \mu_2 \le r) = \mathit{MinEnv}\,(\mu_1 \le r) \sqcap \mathit{MinEnv}\,(\mu_2 \le r) \\
\mathit{MinEnv}\,(\emptyset \le r) = \emptyset & \mathit{MinEnv}\,(\mathsf{CC} \le \mathsf{H}) = \emptyset
\end{array}$$

(Translate-let)

$$\frac{\Delta;\Gamma \vdash_{\mathcal{L}} e_1 \Rightarrow \Delta_1 \vdash e_1' : \underset{\mu_1}{-}\sigma_1 \quad \langle \sigma_1',\ \Delta_2\rangle = \mathit{Close}(\sigma_1, \Delta_1, \Gamma) \quad \Delta_2;\Gamma_x, x:\sigma_1' \vdash_{\mathcal{L}} e_2 \Rightarrow \Delta' \vdash e_2' : \underset{\mu_2}{-}\sigma_2}{\Delta;\Gamma \vdash_{\mathcal{L}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Rightarrow \Delta' \vdash \mathbf{let}^{\mathsf{H}}\ x = e_1'\ \mathbf{in}\ e_2' : \underset{\mu_1 \vee \mu_2}{-}\sigma_2}$$

(Translate-abs)

$$\frac{\sigma = \mathit{Annotate}^{\mathsf{H}}(\tau,\ \mathit{Dom}\,(\Delta)) \quad \Delta;\Gamma_x, x:\sigma \vdash_{\mathcal{L}} e \Rightarrow \Delta' \vdash e' : \underset{\mu}{-}\sigma'}{\Delta;\Gamma \vdash_{\mathcal{L}} \lambda x:\tau.\ e \Rightarrow \Delta' \vdash \langle \lambda^{\mathsf{H}}\, x{:}\sigma.\ e'\rangle^{\mathsf{H}} : \underset{\emptyset}{-}(\sigma \xrightarrow[\mu]{\mathsf{H}} \sigma')}$$

(Translate-callcc)

$$\frac{\Delta;\Gamma \vdash_{HL} e \Rightarrow \Delta' \vdash e' : \underset{\mu}{-}(\sigma\ {}^{\mathsf{H}}\mathsf{cont} \xrightarrow[\mu']{\mathsf{H}} \sigma)}{\Delta;\Gamma \vdash_{HL} \mathbf{callcc}\ e \Rightarrow \Delta' \vdash \mathbf{let}^{\mathsf{H}}\ x = e'\ \mathbf{in}\ \mathbf{callcc}\ x : \underset{\mu \vee \mu' \vee \mathsf{CC}}{-}\sigma}$$

Fig. 5. Typed translation from *HL* to *RL*

Several auxiliary functions are shown in the figure, and the definitions of several simpler functions are as follows. The lub of two resources is defined by $r \sqcup r = r$ and $\mathsf{S} \sqcup \mathsf{H} = \mathsf{H}$. The function $\sqcap$ for merging two effect environments is defined as $(\Delta_1 \sqcap \Delta_2)(t) = \Delta_1(t) \sqcup \Delta_2(t)$ if $t \in Dom(\Delta_1) \cap Dom(\Delta_2)$, and $(\Delta_1 \sqcap \Delta_2)(t) = \Delta_i(t)$ on the rest of $Dom(\Delta_1) \cup Dom(\Delta_2)$. The free effect variables of a type σ are denoted by $fev(\sigma)$; the function $Close(\sigma, \Delta, \Gamma)$ returns the pair $\langle \forall \overline{t_i \leq \Delta(t_i)}.\, \sigma,\ \Delta \backslash_{\overline{\{t_i\}}} \rangle$, where $\overline{\{t_i\}} = fev(\sigma) - fev(\Gamma)$, and similarly we have $CloseAll(\sigma, r) = \forall \overline{t_i \leq r}.\, \sigma$ where $\overline{\{t_i\}} = fev(\sigma)$.

Separately compiled **external** functions are treated as parameters of the compiled *HL* fragment and are wrapped to convert the *HL* resources (continuation heap) to *SL* resources (continuation stack). The wrapping is performed by an auxiliary function invoked as $Wrap_r^{r'}(C, x, \sigma)$, which produces a term coercing x from type σ to type $ConvertType_r^{r'}(\sigma)$ with resource annotations r' in place of r, and places it in context C. When compiling separately, the effects of an **external** function are approximated conservatively by applying Max^r to the effect-annotated declared type of the function; by definition $Max^r(\sigma)$ is $\sigma_1 \xrightarrow[MaxEff(r)]{r} Max^r(\sigma_2)$ when $\sigma = \sigma_1 \xrightarrow[\mu]{r} \sigma_2$, and σ otherwise. This allows the view of external functions as effect-polymorphic without restricting their actual implementations.

6 Related Work and Conclusions

The work presented in this paper is mainly inspired by recent research on effect inference [6, 10, 11, 23, 24], efficient implementation of first-class continuations [2, 8, 22, 1], monads and modular interpreters [30, 12, 29, 13], typed intermediate languages [7, 26, 20, 17, 16, 3], and foreign function call interface [9, 18]. In the following, we briefly explain the relationship of these work with our resource-based approach.

- **Effect systems.** The idea of using effect-based type systems to support language interoperation was first proposed by Gifford and Lucassen [5, 6]. Along this direction, many researchers have worked on various kinds of effect systems and effect inference algorithms [10, 11, 23, 24, 28]. The main novelty of our effect system is that we imposed a "resource-based" upper-bound to the effect variables. Effect variables in all previous effect systems are always *universally* quantified without any upper bounds, so they can be instantiated into any effect expressions. Our system limits the quantification over a finite set of resources—this allows us to take advantage of the effect-resource relationship to support advanced compilation strategies.
- **Efficient call/cc.** Many people have worked on designing various strategies to support efficient implementation of first-class continuations [2, 8, 22, 1]. To support a reasonably efficient call/cc, compilers today mostly use "stack chunks" (a linked list of smaller stacks) [2, 8] or they simply heap allocate all activation records [22]. Both of these representations are incompatible with those used by traditional languages such as C and C++ where activation records are allocated on a sequential stack. First-class continuations thus always impose restrictions and interoperability challenges to the underlying compiler. In fact, many existing compilers choose not to support call/cc, simply because call/cc is not compatible with standard C

calling conventions. The techniques presented in this paper provide opportunities to support both efficient call/cc and interoperability with code that use sequential stacks.

- **Threads.** Implementing threads does not necessarily require first-class continuations but only an equivalent of one-shot continuations [1]. A finer distinction between these classes of continuations is useful, however the issues of incorporating linearity in the type system to ensure safety in the presence of one-shot continuations are beyond the scope of this paper.
- **Monads and modular interpreters.** The idea of using *resources* and *effects* to characterize the run-time configuration of a function is inspired by recent work on monad-based interactions and modular interpreters [30, 12, 29, 13]. Unlike in the monadic approach, our system provides a way of switching the runtime context "horizontally" from one to another via the $\mathbf{use}^r\ (e)$ construct.
- **Typed intermediate languages.** Typed intermediate languages have received much attention lately, especially in the HOT (i.e., higher-order and typed) language community. However, recent work [7, 14, 21, 17, 3, 16, 15] has mostly focused on the theoretical foundations and general language design issues. The type system in this paper focused on the problem of compiling multiple source languages into a common typed intermediate format. We plan to incorporate the resource and effect annotations into our FLINT intermediate language [21].
- **Foreign function call interface.** The interoperability problem addressed in this paper has much in common with frameworks for multi-lingual programming, such as ILU, CORBA [27], and Microsoft's COM [19]. It also relates to the foreign function call interfaces in most existing compilers [9, 18]. Although these work do address many of the low-level problems, such as converting data representations between languages or passing information to remote processes, their implementations do not provide any safety guarantees (or if they do, they would require external programs run in a separate address space). The work presented in this paper focuses on interfacing programs running in the single address space with much higher performance requirements. We emphasize building a *safe*, *efficient*, and *robust* interface across multiple HOT languages.

We believe what we have presented in this paper is a good first-step towards a fully formal investigation on the topic of safe fine-grain language interoperations. We have concentrated on the issues of first-class continuations in this paper, but the framework presented here should also apply to handle other language features such as states, exceptions, and non-termination. The effect system described in this paper is also very general and useful for static program analysis: because it supports effect polymorphism, effect information is accurately propagated through high-order functions. This is clearly much more informative than the single one-bit (or N-bit) information seen in the simple monad-based calculus [16, 25].

There are many hard problems that must be solved in order to support a safe and fine-grained interoperation between ML and safe-C, for instance, the interactions between garbage collection and explicit memory allocation, between type-safe and unsafe language features etc. We plan to pursue these problems in the future.

Acknowledgment

We are grateful to the anonymous referees for their valuable comments.

References

[1] C. Bruggeman, O. Waddell, and K. Dybvig. Representing control in the presence of one-shot continuations. In *Proc. ACM SIGPLAN '96 Conf. on Prog. Lang. Design and Implementation*, pages 99–107, New York, June 1996. ACM Press.

[2] W. D. Clinger, A. H. Hartheimer, and E. M. Ost. Implementation strategies for continuations. In *1988 ACM Conference on Lisp and Functional Programming*, pages 124–131, New York, June 1988. ACM Press.

[3] A. Dimock, R. Muller, F. Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 11–24. ACM Press, June 1997.

[4] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. ACM SIGPLAN '93 Conf. on Prog. Lang. Design and Implementation*, pages 237–247, New York, June 1993. ACM Press.

[5] D. Gifford and J. Lucassen. Integrating functional and imperative programming. In *1986 ACM Conference on Lisp and Functional Programming*, New York, August 1986. ACM Press.

[6] D. K. Gifford *et al*. FX-87 reference manual. Technical Report MIT/LCS/TR-407, M.I.T. Laboratory for Computer Science, September 1987.

[7] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 130–141, New York, Jan 1995. ACM Press.

[8] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proc. ACM SIGPLAN '90 Conf. on Prog. Lang. Design and Implementation*, pages 66–77, New York, 1990. ACM Press.

[9] L. Huelsbergen. A portable C interface for Standard ML of New Jersey. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, January 1996.

[10] P. Jouvelot and D. K. Gifford. Reasoning about continuations with control effects. In *Proc. ACM SIGPLAN '89 Conf. on Prog. Lang. Design and Implementation*, pages 218–226. ACM Press, 1989.

[11] P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Eighteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 303–310, New York, Jan 1991. ACM Press.

[12] J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Proc. ACM SIGPLAN '94 Conf. on Prog. Lang. Design and Implementation*, pages 24–35, New York, June 1994. ACM Press.

[13] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proc. 22rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 333–343. ACM Press, 1995.

[14] G. Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1995. Tech Report CMU-CS-95-226.

[15] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. 25rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, page (to appear). ACM Press, 1998.

[16] S. Peyton Jones, J. Launchbury, M. Shields, and A. Tolmach. Bridging the gulf: a common intermediate language for ML and Haskell. In *Proc. 25rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, page (to appear). ACM Press, 1998.

[17] S. Peyton Jones and E. Meijer. Henk: a typed intermediate language. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[18] S. Peyton Jones, T. Nordin, and A. Reid. Green card: a foreign-language interface for Haskell. Available at http://www.dcs.gla.ac.uk:80/ simonpj/green-card.ps.gz, 1997.

[19] D. Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[20] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[21] Z. Shao. Typed common intermediate format. In *Proc. 1997 USENIX Conference on Domain Specific Languages*, pages 89–102, October 1997.

[22] Z. Shao and A. W. Appel. Space-efficient closure representations. In *1994 ACM Conference on Lisp and Functional Programming*, pages 150–161, New York, June 1994. ACM Press.

[23] J.-P. Talpin and P. Jouvelot. Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 2(3), 1992.

[24] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.

[25] D. Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1996. Tech Report CMU-CS-97-108.

[26] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conf. on Prog. Lang. Design and Implementation*, pages 181–192. ACM Press, 1996.

[27] The Object Management Group. The common object request broker: Architecture and specifications (CORBA). Revision 1.2., Object Management Group (OMG), Framingham, MA, December 1993.

[28] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 188–201. ACM Press, 1994.

[29] P. Wadler. The essence of functional programming (invited talk). In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1992. ACM Press.

[30] P. Wadler. How to declare an imperative (invited talk). In *International Logic Programming Symposium*, Portland, Oregon, December 1995. MIT Press.