Process Migration and Transactions Using a Formal Intermediate Language*

Jason Hickey, Justin D. Smith, Brian Aydemir,
Nathaniel Gray, Adam Granicz, Cristian Ţăpuş
Caltech Computer Science 256-80
Pasadena, CA 91125
{jyh,justins,emre,n8gray,granicz,crt}@cs.caltech.edu

ABSTRACT

Process migration and atomic transactions are essential tools for constructing fault-tolerant distributed systems. Process migration provides location transparency, the ability to perform load-balancing and process checkpointing, and allows processes to be reconstructed after machine failures. Transactions provide fault-isolation by limiting the scope of errors, and permit speculative execution by allowing rollback of overly optimistic computations. We present a compiler that uses a typed intermediate language and a runtime implementation designed to support these services. Our intermediate language is type-safe and general enough to support front-ends for both type-safe and unsafe languages. In addition, our compiler is able to generate code for both ML and ANSI C programs. We include benchmarks that show that our compiler produces programs with competitive performance.

1. INTRODUCTION

The design of software for distributed applications is a challenging task. In addition to the difficulties posed by processor and network failures, distributed applications are often composed of several parts written in different languages. In this paper, we approach these problems by using a typed, semi-functional intermediate language that supports two key features for distributed computing: whole-process migration, and undoable transactions. To address the multi-language issue, the intermediate language is designed to support both type-safe source languages like ML, and unsafe languages like C.

We have implemented our approach in the Mojave system, which currently provides a multi-language compiler for programs written in C, Naml (a language based on Caml-light), and Pascal. Programs in these languages are compiled to a typed "Functional Intermediate Representation" (FIR). Types are maintained throughout the compilation process, which ensures that component interactions respect type and memory safety. The core FIR language is a variant of System F [10] in continuation-passing style (CPS) [1]. In many respects, the FIR language is similar to the typed intermediate language used in the TILT compiler [24, 20]. However, the FIR is much more limited syntactically.

The FIR language has been quite robust as new source languages have been added to the Mojave compiler, but there are tradeoffs, most evident when compiling C. First, the FIR requires that program execution be safe, despite the fact that it is not always possible to maintain exact type information for C. This limitation is resolved using runtime safety checks. Pointers in the runtime are represented with base/offset pairs, requiring more storage and extra computation. Second, the decision to use continuation-passing style means that there is no runtime stack, and C functions allocate storage on the heap, which can be expensive in some cases. However, there are also many advantages. The FIR has made it easy to augment ANSI C with polymorphism and type inference, exceptions, pattern matching, higher-order functions, and safe return of pointers to "automatic" variables.

The paper is organized as follows. In Section 1.1, we discuss related work. The FIR language is the formal foundation we use to specify process migration and transactions. Sections 2 and 3 describe the syntax, type system, and judgments for this language. In Section 4 we describe the four primitives for migration and transactions, and we define their operational semantics. In Section 5 we cover the implementation of these primitives, and in Section 6 we present preliminary benchmarks and performance.

1.1 Related work

The desire for common intermediate languages dates back to the 1950's, significantly with the UNCOL project [18]. More modern versions that support at least part of the goals of multi-language platforms are the Java Virtual Machine [7] and Microsoft's Common Language Runtime [19, 11]. The JVM has many desirable features: it is portable, it is safe, and compiler technology can produce very efficient executables. However, as Meijer points out [19], while the JVM was designed to be generic, it works most effectively for Java. There is little support for (at one extreme) unsafe languages

*A full version of this paper is available as Caltech technical report `caltechCSTR:2002.007` at `mojave.caltech.edu`. This work was supported by the ONR, grant N00014-01-1-0765; DARPA, grant F33615-98-C3613; and AFOSR, grant F49620-01-1-0361.

Submitted to ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI) San Diego, CA.

like C, and (at another extreme) functional languages like ML.

The Microsoft CLR addresses many of these problems, and supports a wide range of languages including parts of C++, OCaml, and Standard ML. However, the CLR does not ensure safety, although Gordon and Syme have been able to demonstrate safety for a substantial fragment [11]. The CLR is a fairly large language. In contrast, the FIR is quite small—and consequently more effort is required from a language front-end to compile to FIR.

Another related area is portable code generators, of which there are several, including MLRISC [9], C-- [17, 16], and `gcc` [23]. As Peyton-Jones suggests, C itself should be considered as a commonly-used assembly language. In each of these cases, the code generator is attacking the problem of portability and machine-code generation, but not the problem of safety.

In the area of C program safety, the Necula et.al. CCured compiler [21] and the Morrisett et.al. Cyclone Safe-C [15] compiler both extend the C language to include extra information needed to infer safety. The systems distinguish between "safe" and "unsafe" pointers, where a safe pointer is always used in ways that can be shown to be safe, and unsafe pointers include all the rest. In both systems, unsafe pointers are represented at runtime by a "fat" pointer that includes safety information, requiring twice the storage of a safe pointer.

CCured extends the type system and uses type inference to determine safety. Cyclone requires explicit annotation by the user: safe and unsafe pointers have different types, and different dereference operators. Compilation is limited to the subset of C that can be proven safe.

In contrast with this work, the Mojave compiler accepts all source files that conform to the ANSI standard, but *all* pointers are represented as fat pointers. We use dead-code elimination coupled with alias analysis to delete provably unnecessary safety information for variables, but this still requires more space than the other two systems. There is no way in Mojave to represent an array of safe pointers; all pointers are 8 bytes on all platforms.

Process migration has been widely studied [25, 6]. Notably, the JoCaml system [5] provides process mobility for OCaml programs based on the join calculus [8]. Our approach to process migration has been heavily influenced by Cardelli's work on the Ambient Calculus [4, 3]; however, our work with whole-process migration is only the first step toward fine-grained mobility.

Transactions are a fundamental concept in the database community, but again, implementations for general-purpose languages are limited. As part of the Venari project, Haines et.al. [13] implement a transaction mechanism as part of Standard ML. Undoability is implemented by extending the mutation log produced by the generational garbage collector. Our approach (described in Section 5.3.2) also uses a mutation log. However, we combine a generational mark-sweep collector with a copy-on-write mechanism to reduce the cost of rollback and commit operations.

2. THE FIR SYNTAX

In the syntax descriptions below, we use the following conventions. In general, we use the meta-variables i and j to refer to arbitrary integers, and the meta-variables m and n to refer to arbitrary nonnegative integers. In most cases,

		Entity	Description
		$v_1, v_2, \ldots$	Variable names
		$tv_1, tv_2, \ldots$	
		$\alpha, \beta, \ldots$	Type variables
i	$::=$	$\ldots \mid -1 \mid 0 \mid 1 \mid \ldots$	Integer constants
s	$::=$	$[i_1^1, i_2^1], \ldots, [i_1^n, i_2^n]$	Integer interval set

Figure 1: FIR base terms

we use the meta-variable m to enumerate type parameters $\alpha_1, \ldots, \alpha_m$, and the meta-variable n to enumerate actual parameters $v_1, \ldots, v_n$.

The meta-variable v refers to program variables. The meta-variables t and u refer to program types, while the Greek letters α, β, γ and the meta-variable tv refer to type variables.

The FIR base terms are shown in Figure 1. MCC supports several forms of numbers, including integers of various signedness and precision, and floating-point values of various precisions. For simplicity, we consider only boxed (tagged) signed integers and unboxed integers.

Sets of integers are used in integer pattern matching expressions. The sets are represented by lists of closed intervals $[i_1, i_2]$.

2.1 FIR type system

The FIR has two classes of types, the basic types, shown in Figure 2, and the type definitions, which are parameterized types of the form $\Lambda\alpha_1, \ldots, \alpha_m.t$.

The type $\mathbb{Z}_{box}$ refers to tagged, signed integers. Native integers are represented using $\mathbb{Z}_{raw}$, and must be boxed when stored into memory. Floating-point values and other numerical precisions are supported by the implementation, but we do not describe them here.

The tuple type $\langle t_1, \ldots, t_n \rangle$ represents a tuple $\langle v_1, \ldots, v_n \rangle$, where each value v_i has type t_i. MCC supports other types of safe data blocks, including arrays and unions, but they are omitted here for simplicity.

The unsafe type **data** represents arbitrary data. Values of type **data** are normally used to represent data aggregates for imperative programming languages, like C, that allow the assignment of values to the data area without regard for the data type. Data areas with the **data** type have no explicit substructure.

The function type $(t_1, \ldots, t_n) \rightarrow t$ includes the functions that return a value of type t, given arguments of types $t_1, \ldots, t_n$.

The type $tv[t_1, \ldots, t_m]$ applies arguments to a type definition. If the definition is a parameterized type $\Lambda\alpha_1, \ldots, \alpha_m.t$, the type $tv[t_1, \ldots, t_m]$ is defined as the type $t[t_1/\alpha_1, \ldots, t_m/\alpha_m]$. For example, in a context containing the definition $\gamma = \Lambda\alpha, \beta.\langle\alpha, \beta\rangle$, the type $\gamma[\mathbb{Z}_{box}, \mathbb{Z}_{raw} \rightarrow \mathbb{Z}_{box}]$ is the same as the type $\langle \mathbb{Z}_{box}, \mathbb{Z}_{raw} \rightarrow \mathbb{Z}_{box} \rangle$.

The universal type $\forall\alpha_1, \ldots, \alpha_m.t$ defines a polymorphic type, where t must be a function type. The existential type $\exists\alpha_1, \ldots, \alpha_m.t$ defines a type abstraction. The values in an existential type have the form $\mathbf{pack}(v, t_1, \ldots, t_m)$, where v has type $t[t_1/\alpha_1, \ldots, t_m/\alpha_m]$. The type projection $v.i$ is used for values having existential type $\exists\alpha_1, \ldots, \alpha_m.t$. If a value $v = \mathbf{pack}(v', t_1, \ldots, t_m)$ has type $\exists\alpha_1, \ldots, \alpha_m.t$, then $v.i$ is equivalent to t_i.

		Type	Description
t	$::=$	$\mathbb{Z}_{box}$	Boxed integers
	$\mid$	$\mathbb{Z}_{raw}$	Native integers
	$\mid$	**void**	Void type
	$\mid$	$\langle t_1, \ldots, t_m \rangle$	Tuple type
	$\mid$	**data**	Unsafe data
	$\mid$	$(t_1, \ldots, t_m) \to t$	Function type
	$\mid$	$\alpha, \beta, \ldots$	Polymorphic type vars
	$\mid$	$tv[t_1, \ldots, t_m]$	Type application
	$\mid$	$\forall \alpha_1, \ldots, \alpha_m.t$	Universal types
	$\mid$	$\exists \alpha_1, \ldots, \alpha_m.t$	Existential types
	$\mid$	$v.i$	Abstract type
tydef	$::=$	$\Lambda \alpha_1, \ldots, \alpha_m.t$	Parameterized types

Figure 2: The FIR type system

		Definition	Description
a	$::=$	$\textbf{int}(i)$	Boxed integers
	$\mid$	$\textbf{rawint}(i)$	Raw integers
	$\mid$	v	Variables
	$\mid$	$\textbf{apply}(a, t_1, \ldots, t_m)$	Type application
	$\mid$	$\textbf{pack}(v, t_1, \ldots, t_m)$	Existential pack
	$\mid$	$\textbf{unpack}(v)$	Existential unpack
	$\mid$	$\mathit{unop}\, a$	Unary operation
	$\mid$	$a_1\ \mathit{binop}\ a_2$	Binary operation
e	$::=$	$\textbf{let}\, v\colon t = a\ \textbf{in}\ e$	Basic operations
	$\mid$	$a(a_1, \ldots, a_n)$	Tail-call
	$\mid$	$\textbf{special}\, \mathit{spec}$	Special tail-call
	$\mid$	$\textbf{match}\, a\ \textbf{with}\ s_i \mapsto e_i^{i \in \{1..n\}}$	Case analysis
	$\mid$	$\textbf{let}\, v = \mathit{alloc}\ \textbf{in}\ e$	Allocation
	$\mid$	$\textbf{let}\, v\colon t = a_1[a_2]\ \textbf{in}\ e$	Load from heap
	$\mid$	$a_1[a_2]\colon t \leftarrow a_3;\ e$	Store into heap

Figure 3: FIR atoms and expressions

2.2 FIR expressions

Expressions in the FIR are divided into two classes: the atoms a and general expressions e shown in Figure 3.

2.2.1 *Atoms*

The atoms a represent values, including numbers, variables, and basic arithmetic. Atoms are functional: apart from arithmetic exceptions[1], the order of atom evaluation does not matter. The atoms include the following.

The boxed integers $\textbf{int}(i)$ have type $\mathbb{Z}_{box}$. The raw integers $\textbf{rawint}(i)$ are native, unboxed integer constants with type $\mathbb{Z}_{raw}$. There are two forms for arithmetic: unary operations $\mathit{unop}\, a$, and binary operations $a_1\ \mathit{binop}\ a_2$. The operators, shown in Figure 4, include the normal operations for arithmetic.

The variables v represent values defined in the program environment, described in Section 3. Variables are immutable: the FIR does not include a variable assignment operation.

There are three kinds of polymorphic operations. The $\textbf{apply}(v, t_1, \ldots, t_m)$ atom is a type application of a polymorphic value v to type arguments $t_1, \ldots, t_m$. For the application to be well-formed, the variable v must have universal type $\forall \alpha_1, \ldots, \alpha_m.u$; the atom has type $u[t_1/\alpha_1, \ldots, t_m/\alpha_m]$. The $\textbf{pack}(v, t_1, \ldots, t_m)$ atom performs type abstraction. It has type $t = \exists \alpha_1, \ldots, \alpha_m.u$ when v has type $u[t_1/\alpha_1, \ldots, t_m/\alpha_m]$. The $\textbf{unpack}(v)$ atom is the elimination form for type abstraction. If v has existential type $\exists \alpha_1, \ldots, \alpha_m.u$, the atom has type $u[v.1/\alpha_1, \ldots, v.m/\alpha_m]$. The types $v.i$ represent the type parameter t_i in the original pack operation.

2.2.2 *Expressions*

The $\textbf{let}\, v\colon t = a\ \textbf{in}\ e$ expression forms a new scope, where the variable v is bound to the value of the atom expression a in the expression e. For the expression to be well-formed, the atom must have type t, and the expression e must be well-formed for an arbitrary value v of type t.

The tail-call $a(a_1, \ldots, a_n)$ represents a function call to the function a,[2] with arguments $a_1, \ldots, a_n$. For the tail-call to be well-formed, the function a must have some type $(u_1, \ldots, u_n) \to \textbf{void}$, and each argument a_i must have type u_i. The return type of the function is the empty type **void**. There is no syntactic mechanism for using the return value of a function, and functions never return.

The special-call $\textbf{special}\, \mathit{spec}$ represents a call for process migration, or one of the atomic transaction operations. The *spec* operations are shown in Figure 4.

The operator $\textbf{migrate}\ [i, a_p, a_o]\ a_f(a_1, \ldots, a_n)$ defines a process migration. The operator $\textbf{atomic}\ a_f(a_c, a_1, \ldots, a_n)$ specifies entry into an atomic transaction. The operator $\textbf{rollback}\ [a_l, a_c]$ is used to abort a transaction. The operator $\textbf{commit}\ [a_l]\ a_f(a_1, \ldots, a_n)$ commits the transaction identified by a_l.

The match statement $\textbf{match}\, a\ \textbf{with}\ s_i \mapsto e_i^{i \in \{1..n\}}$ is a pattern match of an integer against multiple sets. Each match case $s_i \mapsto e_i$ specifies an integer (or raw integer) set s_i and an expression e_i to be evaluated if $a \in s_i$. Evaluation is ordered and total. Evaluation chooses the *first* match that succeeds, and the match statement is well-formed only if there is a match case for any possible value of a.

The aggregate data areas include tuples, arrays, elements in a union type, and raw data. The $\textbf{let}\, v = \mathit{alloc}\ \textbf{in}\ e$ expression allocates a data aggregate, using one of the *alloc* forms shown in Figure 4.

Values are projected from an aggregate data area using the $\textbf{let}\, v\colon t = a_1[a_2]\ \textbf{in}\ e$ expression. For the expression to be well-formed, a_1 must be an aggregate, and a_2 must be a valid index into the aggregate. All fields in aggregates are mutable. The $a_1[a_2]\colon t \leftarrow a_3;\ e$ expression assigns value a_3 to field a_2 in aggregate a_1.

3. JUDGMENTS

All judgments, including type and well-formedness judgments, are defined with respect to an environment Γ, which we also call a *context*. The environment contains both variable declarations of the form $v\colon t$, and variable definitions

[1] A notable arithmetic exception is division by zero.

[2] There is no expression for defining functions. Functions are statically defined as part of the program context, discussed in Section 3.1, and function definitions may not be nested. The function a in a tail-call is always a variable v or a type-application $v[t_1, \ldots, t_m]$ where v is defined in the context.

		Definition	Desc
unop	::=	$-\mid ! \mid \cdots$	Arith
binop	::=	$+ \mid - \mid * \mid / \mid \cdots$	Arith
alloc	::=	$\langle a_1, \ldots, a_n \rangle : t$	Tuple
	$\mid$	$\mathbf{malloc}(a) : t$	Rawdata
spec	::=	$\mathbf{migrate}\ [i, a_p, a_o]\ a_f(a_1, \ldots, a_n)$	Migrate
	$\mid$	$\mathbf{atomic}\ a_f(a_c, a_1, \ldots, a_n)$	Entry
	$\mid$	$\mathbf{rollback}\ [a_l, a_c]$	Rollback
	$\mid$	$\mathbf{commit}\ [a_l]\ a_f(a_1, \ldots, a_n)$	Commit

Figure 4: FIR operators

		Definition	Description
h	::=	$\mathbf{int}(i)$	Boxed integers
	$\mid$	$\mathbf{rawint}(i)$	Raw integers
	$\mid$	v	Variables
b	::=	h	Heap values
	$\mid$	$\Lambda\alpha_1, \ldots, \alpha_m.\lambda v_1, \ldots, v_n.e$	Functions
	$\mid$	$\mathbf{pack}(v, t_1, \ldots, t_m)$	Type packing
	$\mid$	$\langle h_1, \ldots, h_n \rangle$	Tuples
	$\mid$	$\langle \textsc{c} \rangle$	Raw data

Figure 5: Heap and store values

of the form $v\colon t = b$. The declaration $v\colon t$ specifies that a variable v has an unspecified value of type t. The definition $v\colon t = b$ specifies that variable v has the value b, and b has type t.

3.1 Heap and store values

The definitions in the context use values of two sorts: heap values h, and store values b, shown in Figure 5. The *heap values* represent atoms that have been fully evaluated. The *store values* are the values in a program store. These include heap values, functions, "packed" values with existential type, and data in each of the aggregate data types: tuples, arrays, elements of a union type, and rawdata.

Functions are universally quantified, with type parameters $\alpha_1, \ldots, \alpha_m$,[3] and actual parameters $v_1, \ldots, v_n$. Elements of type **data** are represented abstractly using the form $\langle \textsc{c} \rangle$; the elements in the data area are not explicitly described.

3.2 Kinds

The program types are also defined/declared as part of the context Γ. For presentation purposes, we classify the program types with *kinds*, which have the following form.

$$\begin{array}{rcl} k_s & ::= & \omega \mid \Omega \\ k & ::= & \omega^m \to k_s \end{array}$$

The kind k_s classifies the type definitions *tydef* as "small" types ω and "large" untagged types Ω, primarily to support efficient garbage collection. The general kind $k = \omega^m \to k_s$ represents a parameterized type definition *tydef*. The number of parameters m may be any nonnegative integer. If $m = 0$, we often omit the type parameters.

[3]We allow $m = 0$ here; that is, a function may or may not have any type parameters.

		Definition	Description
def	::=	$v\colon t$	Variable declaration
	$\mid$	$v\colon t = b$	Variable definition
	$\mid$	$tv\colon k$	Type declaration
	$\mid$	$tv\colon k = tydef$	Type definition
Γ	::=	ϵ	Empty environment
	$\mid$	Γ, def	Adding a definition

$\Gamma \vdash \diamond$	Context Γ is well-formed
$\Gamma \vdash tydef_1 = tydef_2 : k$	$tydef_1$ and $tydef_2$ are equal
$\Gamma \vdash a : t$	Atom a has type t
$\Gamma \vdash b : t$	Store value b has type t
$\Gamma \vdash e : t$	Program e has type t

Figure 6: Program contexts and judgments

3.3 Contexts and judgments

A program context Γ is defined as a set of mutually-recursive declarations and definitions, as shown in Figure 6. There are two forms of definitions. The type definition $tv\colon k = tydef$ defines a type named tv, having kind k, and value *tydef*. The variable definition $v\colon t = b$ defines a variable named v, with type t and store value b. For each definition form there is a corresponding *declaration* form.

We assume that each variable and type variable in a context is defined/declared at most once, and we use alpha-renaming throughout this paper to rename variables as appropriate.

The judgment $\Gamma \vdash \diamond$ specifies that the context Γ is well-formed. A context is well-formed if all of its declarations and definitions are well-formed. For each declaration $v\colon t$ and definition $v\colon t = b$, the term t must be a well-formed type, and the value b must have type t. Similarly, all type definitions in Γ must be well-formed.

The type system includes an equational theory of types. The judgment $\Gamma \vdash tydef_1 = tydef_2 : k$ is a type definition equality judgment. When the judgment is true, $tydef_1$ and $tydef_2$ have the specified kind, and they are equal. There is no separate membership judgment $\Gamma \vdash tydef : k$.

The judgments $\Gamma \vdash a : t$, $\Gamma \vdash b : t$, $\Gamma \vdash e : t$ express typing relations for labels, atoms, store values, and expressions, respectively.

4. OPERATIONAL SEMANTICS

Evaluation is defined on *programs*, which include three parts: the current environment Γ, a checkpoint environment $\mathcal{C}$, which is an *ordered list* of checkpoints, and an expression e to be evaluated. A checkpoint $\langle \Gamma, f(\diamond, a_1, \ldots, a_n) \rangle$ contains a context Γ, and a function $f(\diamond, a_1, \ldots, a_n)$, where $\diamond$ is a special transaction parameter. The function is called if evaluation is resumed from the checkpoint.

C	::=	$\langle \Gamma, f(\diamond, a_1, \ldots, a_n) \rangle$	Single checkpoint
$\mathcal{C}$	::=	$C_m; \ldots; C_1$	Checkpoint environment

Definition 4.1. Fully-defined contexts

A context Γ is said to be fully-defined *if every variable v in the context is defined with the form $v\colon t = b$ and every type variable tv is defined with the form $tv\colon k = tydef$.*

Definition 4.2. Programs

A program *is either the special term* **error***, or it is a triple* $(\Gamma \mid \mathcal{C} \mid e)$ *that satisfies the following conditions.*

- Γ *is fully-defined and* $\Gamma \vdash e : \mathbf{void}$*,*
- *For each* $\langle \Gamma', f(\diamond, a_1, \ldots, a_n) \rangle \in \mathcal{C}$*, the context* Γ' *is fully-defined, and the judgment* $\Gamma', v_c : \mathbb{Z}_{raw} \vdash f(v_c, a_1, \ldots, a_n) : \mathbf{void}$ *holds.*

Intuitively, the **error** term specifies a runtime error during program evaluation (such as an out-of-bounds array access, or a runtime type error during a subscripting operation).

Evaluation is a relation on programs. The relation

$$(\Gamma \mid \mathcal{C} \mid e) \to (\Gamma' \mid \mathcal{C}' \mid e')$$

specifies that the program $(\Gamma \mid \mathcal{C} \mid e)$ evaluates in one step to the program $(\Gamma' \mid \mathcal{C}' \mid e')$. The relation

$$(\Gamma \mid \mathcal{C} \mid e) \to \mathbf{error}$$

specifies that evaluation of the program $(\Gamma \mid \mathcal{C} \mid e)$ results in a runtime error in one step.

4.1 Special-calls

The operational semantics for special-calls are shown in Figure 7. For this paper, we describe only the rules for special-calls. The complete type system and its operational semantics are described in the technical report [14].

When a process migrates, the entire process (including Γ, $\mathcal{C}$, and the continuation function) migrates to a new location, which is just another runtime environment. The operational definition of migration is transparent by design. The program context *must* not change during process migration. This decouples the process execution from the process location, allowing transparent fault recovery in distributed systems.

In the **migrate** $[j, a_{ptr}, a_{off}]$ $a_{fun}(a_1, \ldots, a_n)$ special-call expression, the atoms a_{ptr} and a_{off} specify a string (as a raw-data block and offset) that describes the migration protocol and target (for example, a machine name). The number j is a unique identifier used by the runtime. Operationally, evaluation of the expression leads to process migration followed by the evaluation of the tail-call $a_{fun}(a_1, \ldots, a_n)$.

Atomic transactions are entered with the special-call **atomic** $a_{fun}(a_{const}, a_1, \ldots, a_n)$. The runtime adds a process checkpoint to the checkpoint environment $\mathcal{C}$. This checkpoint can be restored later if the transaction is aborted. Evaluation proceeds with a tail-call $a_{fun}(a_{const}, a_1, \ldots, a_n)$, and the atomic call is treated identically to this tailcall for typing purposes. For technical reasons, a_{const} must have type $\mathbb{Z}_{raw}$.

The **rollback** $[a_{level}, a_{const}]$ special-call aborts a transaction. It is possible to enter several transactions simultaneously (in the source program, transactions are typically nested). The atom a_{level} is an integer that identifies the atomic level, and a_{const} is a transaction parameter.

When a transaction checkpoint $\langle \Gamma, a_{fun}(\diamond, a_1, \ldots, a_n) \rangle$ is rolled back with the **rollback** $[i, j]$ special-call to level i with transaction parameter j, evaluation proceeds as a tail-call $a_{fun}(j, a_1, \ldots, a_n)$ using the original process context Γ and the truncated checkpoint environment $C_i; \ldots; C_1$. All checkpoints with level higher than i are discarded[4].

[4]The level that was rolled back is re-entered by this primitive; in effect, the state that is restored is the state captured immediately after the level was entered.

Transactions are committed with the special-call **commit** $[i]$ $a_{fun}(a_1, \ldots, a_n)$. Operationally, the checkpoint is deleted from the checkpoint context and evaluation continues with a tail-call to the function $a_{fun}(a_1, \ldots, a_n)$.

The FIR does not syntactically require entry and commit operations to be balanced. Instead, programs that attempt to rollback to or commit a checkpoint that does not exist evaluate to the **error** term.

4.2 Type safety

The typing rules for the FIR are straightforward extensions of the rules for System F. The following two theorems summarize the relation between the operational semantics and program typing.

Theorem 4.1. Preservation *If* $(\Gamma_r \mid \mathcal{C}_r \mid e_r)$ *is a valid program and* $(\Gamma_r \mid \mathcal{C}_r \mid e_r) \to (\Gamma_c \mid \mathcal{C}_c \mid e_c)$*, then* $(\Gamma_c \mid \mathcal{C}_c \mid e_c)$ *is a program.*

Theorem 4.2. Progress *If* $(\Gamma_r \mid \mathcal{C}_r \mid e_r)$ *is a program, and* e_r *is not a value* h*, then there is a program* $(\Gamma_c \mid \mathcal{C}_c \mid e_c)$ *such that* $(\Gamma_r \mid \mathcal{C}_r \mid e_r) \to (\Gamma_c \mid \mathcal{C}_c \mid e_c)$*, or* $(\Gamma_r \mid \mathcal{C}_r \mid e_r) \to$ **error***.*

The proof of preservation is a case analysis on the reduction operator, and the progress proof is by induction on the length of the proof of $\Gamma \vdash e_r : t$. The complete proofs can be found in the technical report [14].

5. RUNTIME IMPLEMENTATION

The FIR is machine-independent, and the Mojave compiler architecture is designed to support multiple back-ends, including both native-code and interpreted runtimes. Object code generation is performed in two stages: the FIR is first translated to a "Machine Intermediate Representation" (MIR), which introduces runtime safety checks in a machine-independent form, and then the final object code is generated for the target architecture from the MIR program. We do not discuss the MIR language in detail here; the language itself is similar to the FIR with a simpler type system, and the process of generating MIR code is a straightforward elaboration of the FIR code.

The runtime implementation manages several tasks, including execution of runtime type-checks for subscript operations, garbage collection, process migration, and atomic transactions. To complicate matters, a faithful C pointer semantics rules out direct use of data relocation (which occurs when a process migrates, or during heap compaction). To address these matters we introduce several auxiliary data structures and invariants.

5.1 Runtime data structures and invariants

The runtime consists of the following parts and invariants.

- A *heap*, containing the data for tuples, arrays, unions, and rawdata. A data value in the heap is called a *block*, and the heap contains multiple (possibly non-contiguous) blocks.
- A *text area*, containing the program code. The text area is immutable at all times except during process migration.
- A set of *registers*. Each variable in the program is assigned to a register. At any time during program

$$(\Gamma \mid \mathcal{C} \mid \textbf{special migrate } [j, h_{ptr}, h_{off}]\ f(h_1, \ldots, h_n)) \to (\Gamma \mid \mathcal{C} \mid f(h_1, \ldots, h_n)) \quad \textsc{Red-SysMigrate}$$

$$\begin{array}{ll} (\Gamma \mid \mathcal{C} \mid \textbf{special atomic } f(h_c, h_1, \ldots, h_n)) \to & \textsc{Red-Atomic} \\ \quad (\Gamma \mid \langle \Gamma, f(\diamond, h_1, \ldots, h_n) \rangle ; \mathcal{C} \mid f(h_c, h_1, \ldots, h_n)) & \end{array}$$

$$\begin{array}{ll} (\Gamma' \mid C_m; \ldots; C_i = \langle \Gamma, f(\diamond, h_1, \ldots, h_n) \rangle ; \ldots; C_1 \mid \textbf{special rollback } [i, j]) \to & \textsc{Red-Atomic-Rollback} \\ \quad (\Gamma \mid C_i; C_{i-1}; \ldots; C_1 \mid f(j, h_1, \ldots, h_n)) & \\ \quad \textbf{when } i \in \{1 \ldots m\} & \end{array}$$

$$\begin{array}{ll} (\Gamma \mid C_m; \ldots; C_1 \mid \textbf{special commit } [i]\ f(h_1, \ldots, h_n)) \to & \textsc{Red-Atomic-Commit} \\ \quad (\Gamma \mid C_m; \ldots; C_{i+1}; C_{i-1}; \ldots; C_1 \mid f(h_1, \ldots, h_n)) & \\ \quad \textbf{when } i \in \{1 \ldots m\} & \end{array}$$

Figure 7: Special-call operational semantics

execution, a register may contain a value in one of several machine types: a pointer into the heap, a function pointer, or a numerical value. The machine type is statically determined from the variable's type in the FIR. Register spills have the same properties as registers.

Invariant: if a register contains a pointer, it contains the address of a block in the heap; if a register contains a function pointer, it contains the address of a function entry point in the text area.

- A *pointer table*, containing pointers to all valid data blocks in the heap.

  Invariant: all non-empty entries in the pointer table contain pointers to valid blocks in the heap, and every block in the heap has an entry in the pointer table.

- A *function table*, containing function pointers to all valid higher-order functions. The function table is immutable, except during process migration.

  Invariant: all entries in the function table contain the address of a function entry point in the text area.

- A *checkpoint* record, containing descriptions of all live program checkpoints. Checkpoints are discussed in Section 5.3.

5.1.1 Data blocks and the heap

The heap represents the FIR store, and it contains the store values b defined in Figure 5, which we call *blocks*. The runtime representation of a block contains two parts: a header that describes the size and type of information stored in the block, and a value area containing the contents of the block.

There are two types of data blocks in the heap. Unsafe data corresponds to values of type **data**. Safe data is type-safe ML data, and corresponds to the $\langle t_1, \ldots, t_n \rangle$, t **array**, and **union**$(tv[t_1, \ldots, t_n], s)$ types.

The contents of unsafe data are not explicitly typed in the FIR, and safety checks are required to ensure the data is interpreted properly. Any pointer read from an unsafe block must be checked to ensure it is a valid pointer, and the bounds must be checked any time an unsafe block is dereferenced. In contrast, the contents of safe block data are typed in the FIR, and many safety checks can be omitted[5]. The garbage collector can use explicit FIR types to identify pointers embedded in safe block data, but it must use a more conservative algorithm to determine which values are pointers in unsafe block data[6].

The pointer table contains the address of each valid live block in the heap. A block header has three parts: it contains 1) a tag that identifies the block type (unsafe or safe), 2) an index into the pointer table identifying the pointer for this block, and 3) a nonnegative number that indicates the size of the block. The tag field is overloaded to indicate the union case for data in a disjoint union. The header also contains bits used by garbage collection and transactions.

5.1.2 Pointer table

The pointer table *ptable* effectively acts as a segment table for the blocks (segments) in the heap. It supports several features, including migration and transactions, but its main purpose is to allow for relocation and safety for C data areas. The pointer table is implemented in software, however its design is compatible with a hardware implementation for increased efficiency.

Figure 8 illustrates the pointer table layout. The pointer table contains entries pointing to all allocated data blocks. Source-level C pointers are represented in the runtime as (base + offset) pairs. The base pointer always points to the beginning of a data block in the heap, and the offset is a signed byte index relative to the base. Base pointers are never stored directly in the heap. Instead, the base pointer is stored as an index to an entry in the pointer table, which contains the actual address of the beginning of the data block.

The pointer table serves several purposes. First, it provides a simple mechanism for identifying and validating data pointers in aggregate blocks. When an index i for a base pointer is read from the heap, the following steps are performed:

1. i is checked against the size of the pointer table to verify if it is a valid index.

2. The value p is read from the i^{th} entry in the pointer table.

[5]Safety checks cannot be omitted on data after a successful migration, unless the two machines are mutually trusting. By default, the destination machine generates safety checks on all data.

[6]As a consequence, the garbage collector may consider certain blocks to be live beyond their real live range.

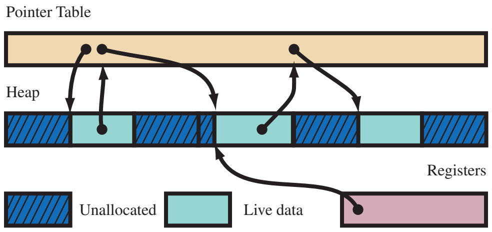

Figure 8: Pointer table representation

3. p is checked to ensure it is not free (that it points into the heap).

After these steps, p is always a valid pointer to the beginning of a block. For additional safety, we can verify that the index stored in p's block header matches index i. These steps can be performed in a small number of assembly instructions, requiring only two branch points.

The second purpose of the pointer table is to support relocation. If the heap is reorganized by garbage collection or process migration, the pointer table (and registers) are updated with the new locations, but the heap values themselves are preserved. This level of transparency has a cost: in addition to the execution overhead, the header of each block in the heap contains an index. In the IA32 runtime, if we include the pointer table overhead, the overhead is in excess of 12 bytes per block.

5.1.3 Function pointers

Function pointers are managed through the function table *funtable* that serves the same purpose as the pointer table for heap data. A function stub must be generated for each FIR function that escapes (higher-order functions). Each function stub has a function header, and the function table contains the addresses of all the escaping-function headers. To ease some of the runtime safety checks, each function stub is formatted with a block header that indicates that the function is a data block with zero-size. As with block pointers, function pointers are represented in the heap as indexes into the function table.

The function header also contains an arity tag, used to describe the types of the arguments. Arity tags are used when a function is called to ensure that a function is called with arguments that are compatible with the function's signature[7]. The arity tags are integer identifiers, computed at link time from the function signatures. The signatures themselves are generated based on the primitive architecture types, not the high-level FIR types[8]. When a function is called, the arguments have the same arity tag as the function signature, or the runtime raises an exception.

5.1.4 Pointer safety

The runtime operations for load $\langle \text{C} \rangle\, [i] : t$ and store $\langle \text{C} \rangle\, [i] : t \leftarrow h$ are guaranteed to be type-safe, even for unsafe blocks. The runtime safety check for a load operation is performed as follows.

1. The index i is compared with the bounds of block $\langle \text{C} \rangle$; an exception is raised if the index is out-of-bounds.
2. The value h at location i is retrieved, and a safety check is performed.
   - If t represents a pointer, then h should be an index into the pointer table. If h is a valid pointer table index, and the entry $\mathit{ptable}[h]$ is a valid pointer p, the result of the load is p.
   - If t represents a function pointer, then h should be an index into the function table. If h is a value function table index, the result of the load is $\mathit{funtable}[h]$.
   - Otherwise, h does not represent a pointer, and the result of the load is h.

The safety check for a store operation is somewhat simpler. For a store operation $\langle \text{C} \rangle\, [i] : t \leftarrow h$, the runtime invariants guarantee that if t represents a pointer, then h is a valid pointer to a block in the heap; and if t is of function type, then h is a valid pointer to a function header. In these two cases (after a bounds-check on the index i) the index for h is stored. If t does not represent either kind of pointer, the value h is stored directly.

5.2 Process migration

In a distributed system, a process will execute on a specific machine with a particular architecture. Since individual nodes in a cluster may fail at any time, a mechanism for migrating a process from one machine to another is an essential tool for fault-tolerance. Such a mechanism needs to perform three operations: a *pack* operation to capture the entire state of the process, including the program counter, all register values, heap data, and code; a *transmit* operation to transmit the state of the process to a target machine; and an *unpack* operation to reconstruct the process state on the target machine and resume execution. Collectively, this sequence of operations is referred to as *process migration*.

Process migration should be architecture-independent, to allow for distributed clusters of heterogeneous nodes. Also, process migration should be safe; the remote machine receiving the program should be able to verify that the program type-checks and that heap values are used in a proper manner. If the remote machine can verify that a received program is safe, then we can use process migration in environments where machines in the cluster do not trust each other entirely, such as the wide-area computing clusters on the Internet.

Note that since process migration requires *pack* and *unpack* operations, it is fairly straightforward to extend the mechanism to support saving the process state to a file for later execution, and to write checkpoint files while the process is running that contain snapshots of the full process state. In the event of a later failure, the process can be recovered from this file using the *unpack* operation.

5.2.1 Runtime support for migration

The implementation of the *pack* and *unpack* operations is relatively straightforward. Since all heap data and function pointers in the heap are represented indirectly as indices, the heap data is not modified by a migration, even if the data are relocated. The *pack* operation first performs garbage collection and then packs the live data in the heap, the pointer

[7] This must be checked at runtime since C permits function pointers to be coerced arbitrarily.

[8] The primitive architecture types are currently `value`, `pointer aggr`, `pointer block`, `poly`, and `function` ◇.

table, the program text, and the registers into a message that can be stored or transmitted.

In order to achieve architecture independence, we never migrate the actual executable text[9]. Instead we migrate the FIR code for the program. The location index i in the migration call is used to correlate the runtime execution point with a corresponding execution point in the FIR. On an *unpack* operation, the FIR code is type-checked, recompiled, and execution is resumed. Note that for C programs, the size and byte-ordering of the various data types must conform to a uniform standard.

It is possible to migrate values stored in hardware registers across architectures. Note that the only live variables across migration are the arguments $(a_1, \ldots, a_n)$ passed to function f. This corresponds exactly to the set of register values which will be live during migration. To migrate these values, the backend packs them into a newly allocated block in the heap, taking care to convert any real pointers into index values. This allows us to use an architecture-dependent representation of values in the registers, and also provides safety checks on register values automatically, when they are read out of the heap on the target machine.

5.3 Atomic operations and transactions

In databases, transactions play a key role in ensuring that sequences of operations that are run simultaneously do not interfere. Semantically, transactional execution appears atomic; that is, either all the operations in a transaction must succeed, or none of them will succeed.

The primary obstacle in implementing atomic transactions is restoration of the program state[10]. When a transaction is aborted, the entire process state, including all variable and heap values, must be restored to the state it had on entry into the transaction.

Rollback can be expressed with process migration by having a process write a new checkpoint file each time it enters a new atomic section. If the transaction is aborted, the previous state can be restored by restoring the process from a checkpoint. However, since the migration mechanism recompiles the program, and the *entire* process state must be reconstructed, this operation can be very expensive. Even taking the checkpoint is expensive, since the entire state must be written to a file, even parts of the state that have not changed since a prior checkpoint. By contrast, atomic transactions use a copy-on-write mechanism to keep track of modified state that must be restored if the transaction is rolled back.

The FIR provides three primitives for managing atomic transactions: *entry*, which enters a new atomic level; *commit*, which marks an atomic level as completed; and *rollback*, which aborts all changes made by a particular level and resumes execution at the point where the level was previously entered.

[9]As a future optimization, we may include support for migrating executable code when the architecture is known. Such an optimization would compromise program safety however, since it is not trivial to verify the correctness of assembly code.

[10]In this paper, we do not consider rollback of I/O operations. However, the concepts discussed here can be extended to include I/O operations with some assistance from the operating system. These extensions will be discussed in a future paper.

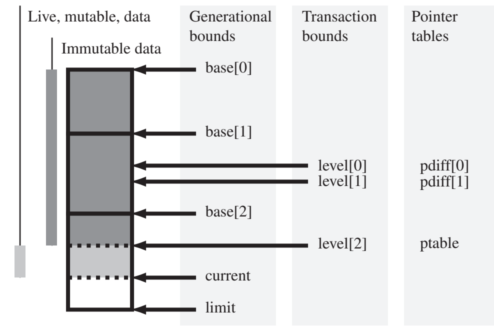

Figure 9: Heap data with multiple atomic levels

5.3.1 *Using atomic transactions in the FIR*

Atomic transactions may be nested; each *entry* operation enters a new atomic level nested within the previous level. Atomic levels are numbered from 1 to N, where 1 is the oldest atomic level entered and N is the most recent. A process that has not entered any atomic transactions is at level 0. A level l keeps track of all changes made to the state that have occurred since l was entered. Atomic levels use copy-on-write semantics; when a block in the heap is modified, the block is cloned and the pointer table updated to point to the new copy of the block, preserving the data in the original block. On a *commit* or *rollback* operation of l, exactly one of these blocks will be discarded.

Rollback resumes execution at the point where level l was entered. No function or argument list is specified; the function that was associated with level l is saved as part of the checkpoint, to be called with the original atom arguments but with the new value for c. This version of the primitive is a retry primitive; atomic level l is automatically re-entered after it (and all later levels) have been rolled back[11].

5.3.2 *Implementation of atomic transactions*

Transactions are implemented in close cooperation with the garbage collector. The heap layout is shown in Figure 9, which is drawn with the base of the heap at the top of the figure, and the limit of the heap at the bottom.

The heap has the following properties.

- Each heap generation i is delimited by a base pointer $base[i]$, and a limit pointer $base[i+1]$, or, in the case of the youngest generation, *limit*.

- The upper bound of atomic level i is delimited by the $level[i]$ pointer.

  Invariant: (ATOMIC INVARIANT) all heap data for atomic level i is between $base[1]$ and $level[i]$, and it is immutable.

The generational bounds and the atomic level bounds are independent. An atomic level may cross a generational boundary, and often does. The two are related however: since the data in an atomic level is immutable, garbage collection on an atomic level is idempotent.

[11]In effect, the state that is captured and restored is the state immediately after level l was entered.

5.3.2.1 *Garbage collection.*

The garbage collector uses generational, mark-sweep, compacting, collection. During the mark phase for generation i, the entire live set for generation i and all younger generations is traversed, and the live blocks are marked. The marking algorithm uses a pointer-reversal scheme to eliminate the need for additional storage during traversal.

During the sweep phase for generation i, the heap from $base[i]$ to $limit$ is scanned. If a dead block is encountered, its entry in the pointer table is deleted. Otherwise, if the block is live, it is copied left to the lowest unallocated location in the heap, and the pointer table is updated with the new location. Note that the heap data itself is not modified during collection.

Block ordering is preserved by the collection, and the heap pointers $level[i]$, $base[i]$, $current$, and $limit$ are updated after an allocation to point to their new locations.

5.3.2.2 *Transaction support.*

On atomic entry, a new generation is set up in the heap by creating a new $level[i]$ pointing at $current$, the end of the heap. All data before $current$ becomes immutable. In addition to the $level[i]$ bounds, each level has its own pointer table $pdiff[i]$, that can be used to restore the pointer table if the transaction is aborted. To minimize storage requirements, the $pdiff[i]$ table is stored as a set of differences with the current pointer table $ptable$.

Within a transaction, the only operations requiring special support are the assignment operations. If a block is to be mutated, and the block belongs to a previous generation (its address is below the current $level[i]$), the block is copied into the minor heap, the current pointer table $ptable$ is updated with the new location, and the previous pointer table $pdiff[i-1]$ is updated with the block's original location. The original data remains unmodified. The garbage collector includes the *pdiff* tables as "root" pointers; the original block remains live.

When an atomic level i is committed, the pointer $level[i]$ and the difference table $pdiff[i]$ are deleted. In general, this will release storage that was needed in case of rollback, and the space is automatically reclaimed during the next garbage collection. On a rollback to atomic level i, the pointer table is restored from the current pointer table and the $pdiff[i]$ table, which is deleted along with the $level[i]$ delimiter.

6. BENCHMARKS

System benchmarks are shown in Figure 10 for version 0.5.0 of the Mojave compiler, which was released in May 2002, about a year after the Mojave project started. The performance numbers measure total real execution time on an unloaded 700MHz Intel Pentium III. The Mojave system is freely available at `mojave.caltech.edu` under the GNU General Public License.

The Mojave system is currently under development, and benchmark performance varies widely. Performance numbers are given for several compilers. The `gcc` column uses the GNU compiler collection, version 2.96; `gcc2` uses the `-O2` optimization. The `mcc2` columns list performance numbers for the Mojave compiler. For comparison purposes (only), the `mcc2u` column lists performance without runtime safety checks. In the current state of development, the `mcc2` compiler performs only minimal optimization, including dead-code elimination, function inlining, and assembly peephole optimization. Advanced FIR optimizations are fairly easy to implement, and the `mcc6u` column lists performance numbers using an optimizer under development that implements alias analysis and partial redundancy elimination. Naml benchmarks are similar, and include numbers for the INRIA OCaml compiler [22], version 3.04.

C benchmarks (time in seconds)					
Name	gcc	gcc2	mcc2	mcc2u	mcc6u
fib 35	1.0	0.78	4.6	4.6	4.32
mandel	54.7	42.1 (5.5)	7.2	7.3	6.0
msort1	3.83	1.15	5.92	3.01	
msort4	5.4	1.15	8.22	4.13	
imat1	37.1	6.27	27.9	17.3	7.6
fmat1	8.9	2.98	10.2	8.33	4.86
migrate			1.77		
regex			2.87		

Naml benchmarks (time in seconds)				
Name	ocamlc	ocamlopt	mcc2	mcc2u
fib 35	3.89	0.61	8.33	7.81
mandel	545	8.1	183	160

Figure 10: Mojave benchmarks

The specific benchmarks include the following. The `fib` program computes the n^{th} Fibonacci number (using the naive algorithm). This benchmark is highly recursive, and the performance numbers reflect the use of continuation-passing style. The `mcc` programs allocate an exponential number of closures *on the heap*, and much of the time is spent in garbage collection.

The `mandel` benchmark computes a Mandelbrot set. This is a special case where `mcc` C compiler, using the standard optimizations, happens to perform significantly better than `gcc -O2` (performance numbers for `gcc -O3` are shown in parentheses). In contrast, the performance for Naml reflects the use of minimal optimization. The program is implemented with fixed-point numbers, and each arithmetic operation is a function call. The `ocamlopt` compiler inlines the function calls, while `mcc2` and `ocamlc` do not.

The `msort` benchmarks implement a bubble-sort algorithm, `imat1` performs integer matrix multiplication, and `fmat1` tests floating-point matrix multiplication.

The `migrate` benchmark measures the "minimal" process migration time. The program consists of a single migration call. Nearly all of the time is spent in recompilation on the target machine.

The `regex` algorithm is a naive, imperative implementation of a Unix-style regular-expression matcher, using transactions to perform backtracking. The time listed is for determining that the pattern `*h*e*l*l*o*w*o*r*l*d*` occurs in the text of the introduction to this paper. The benchmark enters 945341 transactions with a maximum transaction nesting depth of 6833.

7. CONCLUSION

The Mojave compiler is in an early stage of development, but we believe that it demonstrates the feasibility of practical process migration and transactional computing. We intend the Mojave compiler to be a testbed for the development of distributed algorithms, as well as the application of domain-specific formal methods [2, 12]. As future work, we are investigating transactional filesystem support, as well as the implementation of multi-threaded process migration and transactions.

8. REFERENCES

[1] A.W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Brian Aydemir, Adam Granicz, and Jason Hickey. Formal design environments. Appears in the supplemental proceedings of the *15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '02)*, NASA technical report NASA CP-2002-211736.

[3] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 365–377, 2000.

[4] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, pages 198–229, 2001. Special Issue on Coordination.

[5] Sylvain Conchon and Fabrice Le Fessant. Jocaml: mobile agents for Objective-Caml. In *ASA/MA'99 Joint Agents Symposium*, October 1999.

[6] G. Di Marzo Serugendo, M. Muhugusa, and C. Tschudin. A survey of theories for mobile agents. *World Wide Web Journal, special issue on Distributed World Wide Web Processing: Applications and Techniques of Web Agents*, 1998.

[7] J. Engel. *Programming for the Java Virtual Machine.* Addison Wesley, 1999.

[8] Cdric Fournet, Georges Gonthier, Jean-Jacques Lvy, Luc Maranget, and Didier Rmy. The reflexive chemical abstract machine and the join-calculus. In *The 23rd Annual ACM SIGPLAN–SIGACT Symposium on Principles Of Programming Languages (PoPL'96)*, January 1996.

[9] Lal George. ML RISC: Customizable and reusable code generators. www.cs.bell-labs.com/~george, 1996.

[10] J-Y. Girard. Une extension de l'interpretation de Gödel a l'analyse, et sonapplication a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symp.*, pages 63–69. Springer-Verlag, NY, 1971.

[11] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *POPL*, 2001.

[12] Adam Granicz and Jason Hickey. Phobos: A front-end approach to extensible compilers. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS-36)*, 2003.

[13] Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems*, November 1994. Short Communication.

[14] Jason Hickey, Justin D. Smith, Brian Aydemir, Nathaniel Gray, Adam Granicz, and Cristian Tapus. Process migration and transactions using a novel intermediate language. Technical Report caltechCSTR 2002.007, California Institute of Technology, Computer Science, July 2002.

[15] J.G.Morrisett, T.Jim, D.Grossman, M.Hicks, J.Cheney, and Y.Wang. Cyclone: A safe dialect of C. In *Usenix Annual Technical Conference*, 2002.

[16] Simon Peyton Jones, D. Oliva, and T. Nordin. C--: a portable assembly language. In *Proceedings of the 1997 Workshop on Implementing Functional Languages*, 1998.

[17] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In *PPDP*, 1999. Invited talk.

[18] S. Macrakis. From UNCOL to ANDF: Progress in standard intermediate languages. Technical report, Open Software Foundation Research Institute, 1993.

[19] Erik Meijer and John Gough. A technical overview of the common language infrastructure. http://research.microsoft.com/~emeijer.

[20] Greg Morrisett, David Tarditi, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. The TIL/ML compiler: Performance and safety through types. (Workshop on Compiler Support for Systems Software, Tucson, Arizona.), February 1996.

[21] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL02)*, 2002.

[22] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object–oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 40–53, 1997.

[23] Richard M. Stallman. Using and porting GNU CC (version 2.0). Technical report, Free Software Foundation, 1992.

[24] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.

[25] Giovanni Vigna, editor. *Mobile Agents and Security.* Springer, 1999. LNCS 1419.