Type Dispatch for Named Hierarchical Types *

Neal Glew

Department of Computer Science
Cornell University

Abstract

Type dispatch constructs are an important feature of many programming languages. Scheme has predicates for testing the runtime type of a value. Java has a class cast expression and a try statement for switching on an exception's class. Crucial to these mechanisms, in typed languages, is type refinement: The static type system will use type dispatch to refine types in successful branches. Considerable previous work has addressed type case constructs for structural type systems without subtyping, but these do not extend to named type systems with subtyping, as is common in object oriented languages. Previous work on type dispatch in named type systems with subtyping has not addressed its implementation formally.

This paper describes a number of type dispatch constructs that share a common theme: class cast and class case constructs in object oriented languages, ML style exceptions, hierarchical extensible sums, and multimethods. I describe a unifying mechanism, *tagging*, that abstracts the operation of these constructs, and I formalise a small tagging language. After discussing how to implement the tagging language, I present a typed language without type dispatch primitives, and a give a formal translation from the tagging language.

1 Introduction

A number of programming languages include *type dispatch* constructs: conditional constructs that test the runtime type of a value. For example, Scheme [KCR98] has predicates for testing if a value is the empty list, an integer, a pair, and so on. These predicates make Scheme more expressive than it would be without them, and can be used to write polymorphic print and marshalling functions and metaprograms like eval. The intermediate language λ_i^{ML} [HM95, Mor95], used in the TIL compiler [TMC+96], has a case construct similar to Scheme's type predicates. The code $\mathsf{typecase}\ \alpha\ \mathsf{of}\ \alpha_1 \times \alpha_2 \Rightarrow e_1 \mid \alpha_1 \to \alpha_2 \Rightarrow e_2 \mid \cdots$ will execute e_1 if α is a pair type, and execute e_2 if α is a function type. The construct is crucial for implementing specialised data representations in the presence of polymorphism, and for writing polymorphic print and marshalling functions.

Class based object oriented languages have a related but slightly different construct: the ability to dispatch on an object's runtime class. The class from which an object is instantiated is called its runtime class. The static type systems of these languages determine a conservative approximation of the runtime class of each expression: The static class of an expression is always a superclass of the runtime class of any object that expression might evaluate to. In part to make up for this conservatism, these languages allow the programmer to test the runtime class of an object with a class cast or a class case mechanism. For example, in Java [GJS96] the expression `(`c`)`e tests if e's runtime class is a subclass of c, and if not throws an exception. Java also has a class case construct, but only for examining the class of an exception packet. The try statement `try` *blk* `catch (`$classname_1$ x_1`)` blk_1 $\cdots$ `catch (`$classname_n$ x_n`)` blk_n executes *blk*, and if *blk* throws an exception, matches that exception's runtime class against $classname_1$ through $classname_n$. If $classname_i$ is the first matching class, x_i is bound to the exception, and blk_i is executed. The ability to examine runtime classes is crucial to Java's exception mechanism, and is generally useful in a number of other situations.

In typed languages, type refinement is a key property of type dispatch constructs: After dispatching on the runtime type of a value, that value's static type changes to reflect the new type information. For example, in the λ_i^{ML} code:

$$\Lambda\alpha.\lambda x{:}\alpha.\mathsf{typecase}\ \alpha\ \mathsf{of}\ \alpha_1 \times \alpha_2 \Rightarrow e_1 \mid \ldots$$

x initially has static type α, but in expression e_1 it is refined to $\alpha_1 \times \alpha_2$. Similarly in Java, if `Student` is a subclass of `Person` and x is declared to have type `Person`, then in the expression `(Student)x`, while x has static type `Person` the whole expression has the refined type `Student`.

The importance of type dispatch constructs to languages that have them, and the need to implement sound type refinement, justifies seeking formal foundations for them. There are two aspects to the formal analysis of these programming language constructs: First, the constructs themselves should be formalised as primitives in some small language and desired properties proven (*e.g.*, type soundness). Second, the common implementation techniques should be formalised as translations between the above languages and

*This paper is based on work supported in part by the NSF grant CCR-9708915, AFOSR grant F49620-97-1-0013, and ARPA/RADC grant F30602-1-0317. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

To appear in the 1999 ACM SIGPLAN International Conference on Functional Programming, September 1999, Paris, France.

ones without type dispatch primitives, and the translations proven correct.

Previous work on type dispatch in typed languages does exist. Abadi *et al.* [ACPP91] introduce the type dynamic with a type case construct, formalise its semantics and typing rules, and prove soundness. The afore mentioned work on λ_i^{ML} treats a similar construct, also formalising it and proving soundness. Crary, Weirich, and Morrisett [CWM98, CW99] show how to formalise type case in a type erasure interpretation rather than the type passing interpretation used in other works. Their work also addresses the implementation aspect that I mentioned: Their language LX [CW99] has no term level type dispatch primitives, only the necessary typing machinery.

This work on type dynamic and type case does not extend to the case for object oriented languages as it neglects two important features: subtypes and the creation of "new" (*i.e.*, generative) types. For these features, the only previous work is that of Reppy and Riecke [RR96] who describe an extension of SML with objects. Included in their language are hierarchically organised object type constructors, and a case mechanism for dispatching on an object's runtime class. They formalise this construct, and prove it sound. They sketch an implementation, but do not formalise it. This paper formalises a tagging language, which contains a similar construct. The tagging language explains class cast and class constructs, ML style exceptions, and hierarchical extensible sums, and could be used in a typed compilation of multidispatch languages such as Cecil [Cha97].

This paper also formalises a translation of the tagging language into a variant of the typed lambda calculus with some additional typing machinery. Like Crary, Weirich, and Morrisett, I use a type erasure interpretation, and the both the tagging and target languages are eraseable.[1] Correctness theorems for the translation are stated here and proven in a companion technical report [Gle99]. I implemented the ideas in a type directed compiler as part of the TALx86 project [MCG+99]. TALx86 is a typed assembly language, and the key ideas of the target language were used to augment TALx86's type system; these extensions were straightforward. This result is significant for without it, type directed compilers must keep type dispatch as a primitive, and language based security must include type dispatch primitives (as the Java Virtual Machine does).

In the remainder of the paper, I will develop the tagging language and discuss its implementation. I begin by describing in more detail the programming language constructs being addressed. From these constructs I extract a core mechanism, and define a small tagging language that abstracts their fundamental operation. Next, I informally discuss how this language could be implemented and the typing issues that arise. This leads to a formal target language and a formal translation from the tagging language to the target language. Finally, I describe some extensions.

2 Four Type Dispatch Constructs

Consider the following four language constructs:

Class Casting and Class Case: In Java, and in other class based languages, objects are created by instantiating a class, and that class is stored in the object when it is created. Java has a downcasting operation $(c)e$ that evaluates e to an object and then tests to see if that object's class is in the subhierarchy under class c. If so, the cast expression evaluates to the object, but has static type c, which is generally a refinement of e's type. If not, an exception is thrown. More generally, these languages might provide a class case mechanism for testing membership in one of several classes. Java has this operation for the particular case of handling exceptions.

Exceptions: At first glance, ML style exceptions might not seem related to downcasting but, in fact, there is a strong connection. Exception declarations are similar to classes in that they create a new exception name with an associated type. Exception packets, like objects, are created from an exception name, and that name is stored in the packet. Exception matching, then, is like downcasting: Known exception names are compared against the name in an exception packet, and successful comparisons allow access to the carried value at the type of the known exception name. Unlike classes, which are arranged hierarchically, ML style exception names are not hierarchical. On the other hand, Java implements exception packets by using objects, and the declaration of new exception names is achieved by subclassing `throwable`.

Hierarchical Extensible Sums: ML style exceptions are also an example of extensible sums. The exception type is like a global sum type that can be extended by user declarations. Each user-declared exception name is a new branch in the sum. A hierarchical extensible sum allows the sum branches to be arranged in a hierarchy. For example, a programmer might define a hierarchical extensible sum type for the primitives of a compiler intermediate language. She might define a constructor of this sum, `intbin`, for binary integer primitives, and then subconstructors under `intbin` for addition, subtraction, and so on. The intermediate language's type checker could match against `intbin`, since all these primitives have the same type, whereas a code generator would match against the more specific constructors to determine the correct instruction to generate. Reppy and Riecke [RR96] describe hierarhical extensible sums in connection with their class case mechanism, and note how they generalise ML style exceptions. Reppy and Fisher are incorporating a form of hierarchical extensible sums in the language Moby [FR99], a research vehicle for ML2000.

Multimethods: Java has single dispatch: Methods can be thought of as functions that are specialised on their first argument's class. Multimethods (*c.f.* Cecil [Cha97]) are a generalisation of this paradigm: A multimethod is a function that is specialised on any, possibly all, of its arguments' classes. Implementing multimethods requires calling specialised code after determining which specialisation applies. The latter could be implemented by comparing the arguments' runtime classes against patterns of known classes. In a type directed compilation framework, when one of these comparisons succeeds, the types of the arguments must be refined to match the types required by the specialised code. These comparisons are instances of the class case construct described above. Multimethods are similar to Castagna

[1] The *erasure* of a term is obtained by deleting all its typing annotations. A langauge is *eraseable*, or is said to have a *type erasure* interpretation, if its terms behave exactly as their erasures do.

et al.'s overloaded functions [CGL95], except the latter are considered in a structural rather than named typed system.

The core mechanism in all of these examples is a *tagging* mechanism. Exception names, classes, and the constructors of an extensible sum are all examples of *tags* that are placed with or within values. Associated with these tags are types that correspond to the tagged values. The language has a *tag if/case* construct with type refinement in the successful branches. Furthermore, in the case of classes and hierarchical sums, the tags form a *tag hierarchy* and the associated types are in a subtype hierarchy parallel to the tag hierarchy. Usually, the tests of a tag case are not "is tag t_1 equal to tag t_2" but "is tag t_1 in the subhierarchy under tag t_2". I shall call "testing if a tag is under another in the tag hierarchy" a *tag check*. The tagging language described in the next section formalises this core mechanism.

3 A Tagging Language

This section describes a tagging language that abstracts the core operation of the type dispatch constructs described in the previous section. The desired operations are: creating hierarchies of type tags, tagging a value with a tag, and comparing the tag of a tagged value against known tags.

A new tag is created by one of two operations: $\mathsf{newtag}(\tau)$ or $\mathsf{subtag}(\tau, e)$. In both cases the new tag is for tagging values of type τ and has type $\mathsf{tag}(\tau)$. The $\mathsf{newtag}(\cdot)$ form creates a top level tag, and the $\mathsf{subtag}(\cdot, e)$ form creates a subtag of tag e. For example, the ML style exception declaration exception *Failure* of string could be coded in the tagging language as:

$$\mathsf{let}\ \mathit{Failure} = \mathsf{newtag}(\mathsf{string})\ \mathsf{in}$$

Note that for expository purposes, examples will use constructs not in the formal language, such as strings, integers, floating point numbers, and let declarations. For an example of subtags, assume string[10] is the type of strings of length 10, and is a subtype of string. The subexception declaration exception *MyFailure* extends *Failure* of string[10] could be coded as:

$$\mathsf{let}\ \mathit{MyFailure} = \mathsf{subtag}(\mathit{Failure}, \mathsf{string}[10])\ \mathsf{in}$$

Values are tagged with the operation $\mathsf{mktagged}(\langle e_1, e_2\rangle)$ where e_1 is the tag, and e_2 the value to be tagged. The result is a value of type tagged. For example, the creation of an exception packet let ep = *Failure* "unimplemented" could be coded as:

$$\mathsf{let}\ \mathit{ep} = \mathsf{mktagged}(\mathit{Failure}, \text{“unimplemented”})\ \mathsf{in}$$

Tagged values are compared against known tags with the operation $\mathsf{if}\ \mathsf{tagof}(e_1) \le e_2\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$ where e_1 is a tagged value and e_2 is a tag. Informally, the tag in e_1 is extracted and compared, along with all its ancestors in the tag hierarchy, to e_2. If any ancestor is equal to e_2, b_1 is executed with x bound to the value in e_1. Otherwise b_2 is executed. For example, exception matching such as:

```
match ep with
   Failure(x) -> printf "Computation failed: %s" x
 | _ -> printf "Some other exception"
```

could be coded as:

```
if tagof(ep) ≤ Failure then
   x.printf "Computation failed: %s" x
else
   printf "Some other exception"
fi
```

The tagging language has the above operations plus n-tuples and functions with their usual introduction and elimination forms. The syntax is:

$$\begin{array}{lrcl} \text{Types} & \tau, \sigma & ::= & \mathsf{tag}(\tau) \mid \mathsf{tagged} \mid \langle \vec{\tau} \rangle \mid \tau_1 \to \tau_2 \\ \text{Terms} & e, b & ::= & x \mid \mathsf{newtag}(\tau) \mid \mathsf{subtag}(\tau, e) \mid \\ & & & \mathsf{mktagged}(e) \mid \\ & & & \mathsf{if}\ \mathsf{tagof}(e_1) \le e_2\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid \\ & & & \langle \vec{e} \rangle \mid e.i \mid \mathsf{fix}\, f(x:\tau_1):\tau_2.b \mid e_1\ e_2 \end{array}$$

The term $\mathsf{fix}\, f(x{:}\tau_1){:}\tau_2.b$ binds f and x in b, and $\mathsf{if}\ \mathsf{tagof}(e_1) \le e_2\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$ binds x in b_1. I consider syntactic objects equal up to α-equivalence. The capture avoiding substitution of x for y in z is written $z[y := x]$, and simultaneous substitution is written $z[y_1, y_2, := x_1, x_2]$. A sequence of objects from a syntactic class X will be written $\vec{X}$.

The operational semantics is given in Figure 1. The key part of the semantics is modelling the identity of tags. Intuitively, the abstract machine that executes tagging language programs has a memory that stores the identities and details of all tags created in the execution so far. I use a well known technique for modelling memory called an allocation style semantics (*c.f.* Morrisett *et al.* [MFH95]). In this style of semantics, a program state let H in e consists of a heap H that models the memory, and an expression e that is currently being evaluated. Heaps are finite maps from addresses to values stored in the memory, called heap values h. Addresses are usually formalised by an abstract construct called locations, or for the tagging language an abstract construct called tag names might be used. For simplicity, I use variables instead. The tagging language's heap values are tuples $\langle \vec{v} \rangle$[2] and tag definitions (τ, s). The latter consists of the type being tagged τ and the optional supertag s, which is either ϵ for no supertag, or a variable that is the identity of the supertag.

The most interesting reduction rules are the rules for $\mathsf{if}\ \mathsf{tagof}(\cdot) \le \cdot\ \mathsf{then}\ \cdot\ \mathsf{else}\ \cdot\ \mathsf{fi}$. A tag check of x against y is formalised by the predicate $\mathit{tagchk}^H(x, y)$ where H is the heap containing the tag definitions, x is the address of the unknown tag, and y is the address of the known tag. The definition of this predicate says that either x and y are the same tag or x has a supertag and the predicate holds for the supertag. The recursiveness of this definition deserves further comment. As tag hierarchies are acyclic, I intend heaps to be nonrecursive (that is, a heap value can only refer to variables defined earlier in the heap). The operational semantics creates heap values that are nonrecursive, and the typing rules (discussed shortly) force typeable heaps to be nonrecursive. Given that heaps are nonrecursive, $\mathit{tagchk}^H(x, y)$ should be considered an inductive definition. If cyclic heaps are considered, we would need to decide what cycles in the tag hierarchy mean, and adjust the definition of $\mathit{tagchk}^{\cdot}(\cdot, \cdot)$ accordingly. The rest of the semantics is fairly standard for

[2] Tuples are allocated in the heap to make the tagging language and the target language closer. This makes it easier to prove operational correctness of the translation.

Syntax:

Values	v	::=	$x \mid \mathsf{mktagged}(v) \mid \mathsf{fix}\, f(x{:}\tau_1){:}\tau_2.b$
Contexts	E	::=	$[] \mid \mathsf{subtag}(\tau, E) \mid \mathsf{mktagged}(E) \mid \langle \vec{v}, E, \vec{e} \rangle \mid E.i \mid E\ e \mid v\ E \mid$ $\mathsf{if}\ \mathsf{tagof}(E) \leq e_2\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid \mathsf{if}\ \mathsf{tagof}(v) \leq E\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$
Heap Values	h	::=	$(\tau, \epsilon) \mid (\tau, x) \mid \langle \vec{v} \rangle$
Heaps	H	::=	$x_1 = h_1, \ldots, x_n = h_n$
Program States	P	::=	$\mathsf{let}\ H\ \mathsf{in}\ e$

Reduction Rules:

$$\mathsf{let}\ H\ \mathsf{in}\ E[I] \mapsto \mathsf{let}\ H'\ \mathsf{in}\ E[e]$$

I	e	H'	Side Conditions
$\mathsf{newtag}(\tau)$	x	$H, x = (\tau, \epsilon)$	x fresh
$\mathsf{subtag}(\tau, y)$	x	$H, x = (\tau, y)$	x fresh
$\langle \vec{v} \rangle$	x	$H, x = I$	x fresh
$x.i$	v_i	H	$H(x) = \langle v_1, \ldots, v_n \rangle; 1 \leq i \leq n$
$v_1\ v_2$	$b[f, x := v_1, v_2]$	H	$v_1 = \mathsf{fix}\, f(x{:}\tau_1){:}\tau_2.b$
$\mathsf{if}\ \mathsf{tagof}(\mathsf{mktagged}(x)) \leq y$ $\mathsf{then}\ z.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$	$b_1[z := v]$	H	$H(x) = \langle x', v, \vec{v} \rangle; tagchk^H(x', y)$
$\mathsf{if}\ \mathsf{tagof}(\mathsf{mktagged}(x)) \leq y$ $\mathsf{then}\ z.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$	b_2	H	$H(x) = \langle x', v, \vec{v} \rangle;$ not $tagchk^H(x', y)$

Tag checking:

$$tagchk^H(x, y) \stackrel{\text{def}}{=} (x = y) \vee (H(x) = (\tau, x') \wedge tagchk^H(x', y))$$

Figure 1: Source Language Operational Semantics

a context and substitution based reduction semantics. Note that the semantics is deterministic, call by value, and left to right.

The tagging language is eraseable, and the types τ in the operations $\mathsf{newtag}(\tau)$ and $\mathsf{subtag}(e, \tau)$ and the tag definitions (τ, ϵ) and (τ, x) are not needed at runtime.

The typing rules appear in Figure 2, and consist of judgements for subtyping $\vdash_{\mathcal{S}} \tau_1 \leq \tau_2$ and for typing expressions $\Gamma \vdash_{\mathcal{S}} e : \tau$, heap values $\Gamma \vdash_{\mathcal{S}} h : \tau$, heaps $\vdash_{\mathcal{S}} H : \Gamma$, and program states $\vdash_{\mathcal{S}} P : \tau$. Here Γ is a typing context that lists the types of variables. Subtyping for tag and tagged types is trivial, and subtyping for tuples and functions is standard. Note that reflexivity and transitivity of subtyping is derivable from the rules given. The rule for subtags requires that e be a tag for type τ' and that τ be a subtype of τ'. The latter ensures that types associated with tags form a subtype hierarchy in parallel to the tag hierarchy. The rule for $\mathsf{mktagged}(e)$ requires that e be a pair of a type tag for τ and a value of type τ. The rule for $\mathsf{if}\ \mathsf{tagof}(e_1) \leq e_2\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$ requires e_1 to have type tagged, e_2 to be a tag for some type σ, b_1 to type check in a context with x of type σ, and b_2 to type check.

The typing rules are sound with respect to the operational semantics. The proof uses the standard techniques, and the only interesting case is in the type preservation of a successful tag comparison. In that case, a tag of type $\mathsf{tag}(\sigma_1)$ is compared against one of type $\mathsf{tag}(\sigma_2)$. If the comparison succeeds, the next program state has the form $b_1[z := v]$ where b_1 has the desired type if z has type σ_2. However, v has type σ_1, so we need to show that $\vdash_{\mathcal{S}} \sigma_1 \leq \sigma_2$, which follows from this lemma:

Lemma 3.1 *If* $\vdash_{\mathcal{S}} H : \Gamma$, $\Gamma \vdash_{\mathcal{S}} x : \mathsf{tag}(\sigma_1)$, $\Gamma \vdash_{\mathcal{S}} y : \mathsf{tag}(\sigma_2)$, *and* $tagchk^H(x, y)$ *then* $\vdash_{\mathcal{S}} \sigma_1 \leq \sigma_2$.

4 Implementation

Real machine languages do not provide primitives like $\mathsf{newtag}(\tau)$, $\mathsf{subtag}(\tau, e)$, $\mathsf{mktagged}(e)$, and $\mathsf{if}\ \mathsf{tagof}(e_1) \leq e_2\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$. Compiler writers must select data structures to represent the tags and algorithms to implement tag checks. The goal of this section is to formalise a typed translation of the tagging language to another language without the above primitives. For now, think of the target of this translation as a typical lambda calculus with physical pointer equality and some typing machinery that I will develop in this section. This typing machinery is general enough to type other less naive strategies as I sketch in Section 5, and can be added to other low level languages such as Typed Assembly Language [MWCG98, MCG+99].

Consider first how a compiler would translate the examples in the previous section ignoring types. To create the new tag *Failure* the compiler would dynamically allocate a new block of memory. Since, throughout the lifetime of this new block, its address is different from the address of any other dynamically allocated memory block, the compiler can use this address as a unique identifier for the tag. The compiler needs to record the position of *Failure* in the tag hierarchy, so it stores a null pointer into the newly allocated block to indicate that *Failure* is at the top level of the hierarchy. I will use ML's none constructor as to represent the null pointer, and ML's $\mathsf{some}(v)$ to represent a non-null pointer to v. Similarly, to create $MyFailure$ the compiler would allocate a new block of memory and store $\mathsf{some}(Failure)$ in it.

To create the tagged value *ep* the compiler would create a pair consisting of *Failure* and the literal string. So the first

Typing contexts Γ are lists $x_1 : \tau_1, \ldots, x_n : \tau_n$.

$\boxed{\vdash_{\mathcal{S}} \tau_1 \leq \tau_2}$

$$\frac{}{\vdash_{\mathcal{S}} \mathsf{tag}(\tau) \leq \mathsf{tag}(\tau)} \qquad \frac{}{\vdash_{\mathcal{S}} \mathsf{tagged} \leq \mathsf{tagged}}$$

$$\frac{\vdash_{\mathcal{S}} \tau_i \leq \sigma_i}{\vdash_{\mathcal{S}} \langle \tau_1, \ldots, \tau_m \rangle \leq \langle \sigma_1, \ldots, \sigma_n \rangle} \ (m \geq n) \qquad \frac{\vdash_{\mathcal{S}} \tau_2 \leq \tau_1 \quad \vdash_{\mathcal{S}} \sigma_1 \leq \sigma_2}{\vdash_{\mathcal{S}} \tau_1 \to \sigma_1 \leq \tau_2 \to \sigma_2}$$

$\boxed{\Gamma \vdash_{\mathcal{S}} e : \tau}$

$$\frac{\Gamma \vdash_{\mathcal{S}} e : \tau_1 \quad \vdash_{\mathcal{S}} \tau_1 \leq \tau_2}{\Gamma \vdash_{\mathcal{S}} e : \tau_2} \qquad \frac{}{\Gamma \vdash_{\mathcal{S}} x : \tau} \ (\Gamma(x) = \tau)$$

$$\frac{}{\Gamma \vdash_{\mathcal{S}} \mathsf{newtag}(\tau) : \mathsf{tag}(\tau)} \qquad \frac{\Gamma \vdash_{\mathcal{S}} e : \mathsf{tag}(\tau') \quad \vdash_{\mathcal{S}} \tau \leq \tau'}{\Gamma \vdash_{\mathcal{S}} \mathsf{subtag}(\tau, e) : \mathsf{tag}(\tau)} \qquad \frac{\Gamma \vdash_{\mathcal{S}} e : \langle \mathsf{tag}(\tau), \tau \rangle}{\Gamma \vdash_{\mathcal{S}} \mathsf{mktagged}(e) : \mathsf{tagged}}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_1 : \mathsf{tagged} \quad \Gamma \vdash_{\mathcal{S}} e_2 : \mathsf{tag}(\sigma) \quad \Gamma, x : \sigma \vdash_{\mathcal{S}} b_1 : \tau \quad \Gamma \vdash_{\mathcal{S}} b_2 : \tau}{\Gamma \vdash_{\mathcal{S}} \mathsf{if\ tagof}(e_1) \leq e_2 \ \mathsf{then}\ x.b_1 \ \mathsf{else}\ b_2 \ \mathsf{fi} : \tau} \ (x \notin \mathrm{dom}(\Gamma))$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_i : \tau_i}{\Gamma \vdash_{\mathcal{S}} \langle e_1, \ldots, e_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle} \qquad \frac{\Gamma \vdash_{\mathcal{S}} e : \langle \tau_1, \ldots, \tau_n \rangle}{\Gamma \vdash_{\mathcal{S}} e.i : \tau_i} \ (1 \leq i \leq n)$$

$$\frac{\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash_{\mathcal{S}} b : \tau_2}{\Gamma \vdash_{\mathcal{S}} \mathsf{fix}\, f(x{:}\tau_1){:}\tau_2.b : \tau_1 \to \tau_2} \ (f, x \notin \mathrm{dom}(\Gamma)) \qquad \frac{\Gamma \vdash_{\mathcal{S}} e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash_{\mathcal{S}} e_2 : \tau_1}{\Gamma \vdash_{\mathcal{S}} e_1\ e_2 : \tau_2}$$

$\boxed{\Gamma \vdash_{\mathcal{S}} h : \tau \quad \vdash_{\mathcal{S}} H : \Gamma \quad \vdash_{\mathcal{S}} P : \tau}$

$$\frac{}{\Gamma \vdash_{\mathcal{S}} (\tau, \epsilon) : \mathsf{tag}(\tau)} \qquad \frac{\Gamma \vdash_{\mathcal{S}} x : \mathsf{tag}(\tau') \quad \vdash_{\mathcal{S}} \tau \leq \tau'}{\Gamma \vdash_{\mathcal{S}} (\tau, x) : \mathsf{tag}(\tau)} \qquad \frac{\Gamma \vdash_{\mathcal{S}} v_i : \tau_i}{\Gamma \vdash_{\mathcal{S}} \langle v_1, \ldots, v_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle}$$

$$\frac{\vdash_{\mathcal{S}} h_1 : \tau_1 \quad \ldots \quad x_1 : \tau_1, \ldots, x_{n-1} : \tau_{n-1} \vdash_{\mathcal{S}} h_n : \tau_n}{\vdash_{\mathcal{S}} x_1 = h_1, \ldots, x_n = h_n : x_1 : \tau_1, \ldots, x_n : \tau_n} \ (x_1, \ldots, x_n \text{ distinct})$$

$$\frac{\vdash_{\mathcal{S}} H : \Gamma \quad \Gamma \vdash_{\mathcal{S}} e : \tau}{\vdash_{\mathcal{S}} \mathsf{let}\ H\ \mathsf{in}\ e : \tau}$$

Figure 2: Source Language Typing Rules

three examples might become:

$$\begin{array}{l}\mathsf{let}\ \mathit{Failure} = \langle \mathsf{none} \rangle\ \mathsf{in} \\ \mathsf{let}\ \mathit{MyFailure} = \langle \mathsf{some}(\mathit{Failure}) \rangle\ \mathsf{in} \\ \mathsf{let}\ \mathit{ep} = \langle \mathit{Failure}, \text{“unimplemented”} \rangle\ \mathsf{in}\end{array}$$

To translate the last example, the compiler would extract the tag in *ep*, which is the address of some dynamic memory block, and compare it against *Failure*. If they are equal x would be bound to the second component of *ep* and b_1 executed. Otherwise, the supertag would be extracted and the process repeated until there is no supertag, in which case b_2 would be executed. The translated code might be:

$$\begin{array}{l}\mathsf{let}\ z = \mathit{ep}.1\ \mathsf{in} \\ \mathit{loop}1 : \mathsf{if}\ z = \mathit{Failure}\ \mathsf{then\ let}\ x = \mathit{ep}.2\ \mathsf{in}\ b_1 \\ \qquad \mathsf{else\ match}\ z.1\ \mathsf{with\ none} \to b_2 \\ \qquad\quad \mid \mathsf{some}(z') \to (z := z'; \mathsf{goto}\ \mathit{loop}1)\end{array}$$

Now consider designing a type system to annotate the code above. The key difficulty is giving x the correct type. In general, the type of a tagged value like *ep* is unknown, yet if the comparison $z = \mathit{Failure}$ succeeds, the type of *ep*.2 is string, and this fact is needed to give x type string. What makes this difficult is that z and *Failure* are values unrelated to *ep*.2. In order to make this connection clear, the target type system needs to do two things: First, it needs to generate type equalities from physical pointer comparisons; second, it needs to link z and *ep* together, so that type information generated by the comparison will change *ep*'s type.

The solution to the second problem is to use type variables to link tags to values being tagged. For example, *ep*'s type could be $\langle \mathit{tag}(\alpha), \alpha \rangle$ for a yet to be determined type constructor $\mathit{tag}(\cdot)$ and some type variable α. Then comparing z, which has type $\mathit{tag}(\alpha)$, to *Failure*, which has type $\mathit{tag}(\mathsf{string})$, will cause the type checker to change α to string in the successful branch, thus changing *ep*.2's type to string also.

The solution to the first problem lies in the following observation. The compiler is using the address of the memory block allocated for *Failure* as a name for the type string. It never uses the same address as a name for two different types, so if two addresses are equal, the types they name must be equal. To reflect this behaviour the target type system must track which addresses are names for types, and which types they name. The compiler gives the target language type $\mathsf{tag}(\tau_t, \tau_{ds})$ to these pointers. These types resemble the tagging language type $\mathsf{tag}(\tau_t)$, as both are for tags for type τ_t. However, whereas tags were primitives in the tagging language, they are explicit datastructures in the target language, and τ_{ds} reflects the type of these datastructures. In particular, a value v is in the type $\mathsf{tag}(\tau_t, \tau_{ds})$ if

$$
\begin{array}{lrcl}
\text{Types} & \tau, \sigma & ::= & \alpha \mid \mathsf{tag}^{\phi}(\tau_1, \tau_2) \mid \mathsf{rec}\ \alpha.\tau \mid \exists\alpha.\tau \mid \tau? \mid \langle \vec{\tau} \rangle \mid \tau_1 \to \tau_2 \\
\text{Variances} & \phi & ::= & + \mid - \mid \circ \\
\text{Terms} & e, b & ::= & x \mid \mathsf{mktag}(\tau, \langle \vec{e} \rangle) \mid \mathsf{if}\ e_1 = e_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid \mathsf{roll}^{\tau}(e) \mid \mathsf{unroll}(e) \mid \\
 & & & \mathsf{pack}[\tau_1, e]\ \mathsf{as}\ \tau_2 \mid \mathsf{unpack}[\alpha, x] = e_1\ \mathsf{in}\ e_2 \mid \mathsf{none}^{\tau} \mid \mathsf{some}(e) \mid \mathsf{if}?\ e_1\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid \\
 & & & \langle \vec{e} \rangle \mid e.i \mid \mathsf{fix}\, f(x : \tau_1) : \tau_2.b \mid e_1\ e_2 \mid \mathsf{let}\ x_1 = e_1\ \mathsf{and}\ x_2 = e_2\ \mathsf{in}\ e
\end{array}
$$

Figure 3: Target Language Syntax

v also has type τ_{ds} and the programmer declared v to be a tag for type τ_t.

Now consider the datastructures used, and the type τ_{ds} for the examples. *Failure* is a linked list of *Failure*'s ancestors, and each pointer in this linked list is being used as a name for the type string. Linked lists have type $\mathsf{rec}\ \beta.\langle\beta?\rangle$ (where $\tau?$ is an option type). So *Failure* has type:

$$tag(\mathsf{string}) = \mathsf{rec}\ \beta.\mathsf{tag}(\mathsf{string}, \langle\beta?\rangle)$$

The tagged value ep is a pair of such a tag and a string except that string is abstracted over, thus ep has type $\exists\alpha.\langle tag(\alpha), \alpha\rangle$. To get the initial value of z, ep is unpacked introducing α into the type context, and giving z the type $tag(\alpha)$. If z = *Failure* succeeds then the type z tags and the type *Failure* tags must be the same, that is, α is string. The target type system will use this in type checking $\mathsf{let}\ x = ep.2\ \mathsf{in}\ b_1$. Since $ep.2$ has type α, which is string, x will get type the correct type.

Two complications arise with this basic scheme. The first is ensuring that a pointer is used to name only one type. If the same pointer is used to name two different types, runtime type errors could occur. To see this, assume an operation $\mathsf{mktag}(\tau, e)$ that declares that value e is a tag for τ, and consider the following malicious code:

$$
\begin{array}{l}
\mathsf{let}\ x_1 = \langle \mathsf{none} \rangle\ \mathsf{in} \\
\mathsf{let}\ x_2 = \mathsf{mktag}(\mathsf{string}, x_1)\ \mathsf{in} \\
\mathsf{let}\ x_3 = \mathsf{mktag}(\mathsf{float}, x_1)\ \mathsf{in} \\
\mathsf{let}\ y = \langle x_2, \text{“hello”} \rangle\ \mathsf{in}
\end{array}
$$

The variables x_1, x_2, and x_3 are all bound to the same pointer, which points to a tuple with a single element none. However, the type system types x_2 as a tag for strings and x_3 as a tag for floats. The code uses x_2 to created a tagged “hello” value, which is bound to y. Now consider the following innocent code:

$$
\begin{array}{l}
\mathsf{fun}\ foo[\alpha](z : \langle \mathsf{rec}\ \beta.\mathsf{tag}(\alpha, \langle\beta?\rangle), \alpha\rangle) = \\
\quad \mathsf{if}\ z.1 = x_3\ \mathsf{then}\ sin(z.2)\ \mathsf{else}\ 1.0\ \mathsf{fi}
\end{array}
$$

The body of *foo* compares $z.1$ a tag for α to x_3 a tag for float. Therefore in the then branch $z.2$ is refined to type float and the sine computation type checks. However, suppose *foo* was applied to string and y. Since $y.1$ is x_2 which equals x_3, the then branch is executed. But $z.2$ is a string and the sine computation fails. The target type system must ensure that x_1 can be declared a tag for at most one type.

One way to ensure a value is declared a tag for at most one type, is to use a linear type system. If v is of linear type τ^1, then v can be “used” only once. Then it is sufficient for $\mathsf{mktag}(\tau, e)$ to require $e : \sigma^1$ for some σ. However, this requires all the machinery of linear type systems in the target language. A simpler solution, pursued in this paper, is to allow $\mathsf{mktag}(\tau, e)$ only at points where new heap values are created. For example, $\langle\vec{e}\rangle$ creates a new heap value; the target operation $\mathsf{mktag}(\tau, \langle\vec{e}\rangle)$ does the same thing but gives the result type $\mathsf{tag}(\tau, \langle\vec{\tau}\rangle)$ where $\vec{e} : \vec{\tau}$.

The other complication concerns the interaction between subtyping and tag types. In particular, if $\mathsf{tag}(\tau_1, \ldots) \leq \mathsf{tag}(\tau_2, \ldots)$ then what should be the relationship between τ_1 and τ_2 (the second position is covariant, *i.e.*, $\tau_1 \leq \tau_2$ implies $\mathsf{tag}(\tau, \tau_1) \leq \mathsf{tag}(\tau, \tau_2)$). As we shall see, different and conflicting relationships are required by the process of creating subtags and the process of destructing tagged values. The solution is to introduce a variance mechanism that states the relationship that holds. First some terminology, the value $\mathsf{mktag}(\tau, e)$ is said to have been created as a tag for τ. For example, *Failure* was created as a tag for string.

Now consider the creation of a subtag. For this example, string[10] will be the type of strings of length 10, and is a subtype of string. Joe User desires a subtag of *Failure*, *MyFailure*, that tags string[10]. Using the scheme above, the translated code for *MyFailure* is:

$$\mathsf{let}\ \mathit{MyFailure} = \mathsf{mktag}(\mathsf{string}[10], \langle \mathsf{some}(\mathit{Failure}) \rangle)\ \mathsf{in}$$

Consider how this type checks. *MyFailure* should have have $tag(\mathsf{string}[10])$, and the right hand side has this type if *Failure* has type $tag(\mathsf{string}[10])$. In fact, *Failure* has type $tag(\mathsf{string})$ so we require that $tag(\mathsf{string}) \leq tag(\mathsf{string}[10])$, that is:

$$
\begin{array}{rl}
 & \mathsf{rec}\ \beta.\mathsf{tag}(\mathsf{string}, \langle\beta?\rangle) \\
\leq & \mathsf{rec}\ \beta.\mathsf{tag}(\mathsf{string}[10], \langle\beta?\rangle)
\end{array}
$$

This would hold if $\mathsf{tag}(\mathsf{string}, \ldots) \leq \mathsf{tag}(\mathsf{string}[10], \ldots)$, in other words, a contravariant subtyping rule: if $\tau_1 \leq \tau_2$ then $\mathsf{tag}(\tau_2, \ldots) \leq \mathsf{tag}(\tau_1, \ldots)$.

However, now consider tagged value destruction, and the code from above:

$$
\begin{array}{l}
z : \mathsf{rec}\ \beta.\mathsf{tag}(\alpha, \langle\beta?\rangle), ep : \langle \mathsf{rec}\ \beta.\mathsf{tag}(\alpha, \langle\beta?\rangle), \alpha\rangle \\
\mathsf{if}\ z = \mathit{Failure}\ \mathsf{then}\ \mathsf{let}\ x = ep.2\ \mathsf{in}\ b_1\ \mathsf{else}\ \ldots\ \mathsf{fi}
\end{array}
$$

Under the contravariant rule *Failure* subsumes to a tag type for string[10], so the type system could type check $\mathsf{let}\ x = ep.2\ \mathsf{in}\ b_1$ under the assumption that α is string[10]. Under this incorrect assumption, x has type string[10], but $ep.2$ is actually a thirteen character string. Thus for destruction, subtyping must not be contravariant.

I introduce a variance mechanism[3] to track the subtyping rules used. Now a tag type has the form $\mathsf{tag}^{\phi}(\tau_t, \tau_{ds})$ where ϕ is a variance: either covariant $+$, contravariant $-$, or invariant $\circ$. A value is in this type if it has type τ_{ds} and was created to tag type σ. Furthermore, the variance states the relationship between τ_t and σ. For covariance τ_t is a supertype of σ, for contravariant τ_t is a subtype of σ,

[3] Abadi and Cardelli [AC96] describe variances and their typing rules in detail.

Syntax:

Values	v	::=	$x \mid \mathsf{roll}^\tau(v) \mid \mathsf{pack}[\tau_1, v]$ as $\tau_2 \mid \mathsf{none}^\tau \mid \mathsf{some}(v) \mid \mathsf{fix}\, f(x:\tau_1):\tau_2.b$
Contexts	E	::=	$[] \mid \mathsf{mktag}(\tau, \langle \vec{v}, E, \vec{e}\rangle) \mid$ if $E = e$ then b_1 else b_2 fi $\mid$ if $v = E$ then b_1 else b_2 fi $\mid \mathsf{roll}^\tau(E) \mid \mathsf{unroll}(E) \mid \mathsf{pack}[\tau_1, E]$ as $\tau_2 \mid$ $\mathsf{unpack}[\alpha, x] = E$ in $e \mid \mathsf{some}(E) \mid$ if? E then $x.b_1$ else b_2 fi $\mid$ $\langle \vec{v}, E, \vec{e}\rangle \mid E.i \mid E\ e \mid v\ E \mid$ let $x_1 = E$ and $x_2 = e_2$ in $e \mid$ let $x_1 = v$ and $x_2 = E$ in e
Heap Values	h	::=	$\langle \vec{v}\rangle \mid \mathsf{mktag}(\tau, \langle \vec{v}\rangle)$
Heaps	H	::=	$x_1 = h_1, \ldots, x_n = h_n$
Programs	P	::=	let H in e

Reduction Rules:

$$\text{let } H \text{ in } E[I] \mapsto \text{let } H' \text{ in } E[e]$$

I	e	H'	Side Conditions
$\langle \vec{v}\rangle$ or $\mathsf{mktag}(\tau, \langle \vec{v}\rangle)$	x	$H, x = I$	x fresh
if $x = x$ then b_1 else b_2 fi	b_1	H	
if $x = y$ then b_1 else b_2 fi	b_2	H	$x \neq y$
$\mathsf{unroll}(\mathsf{roll}^\tau(v))$	v	H	
$\mathsf{unpack}[\alpha, x] = \mathsf{pack}[\tau_1, v]$ as τ_2 in e	$e[\alpha := \tau_1, x := v]$	H	
if? none^τ then $x.b_1$ else b_2 fi	b_2	H	
if? $\mathsf{some}(v)$ then $x.b_1$ else b_2 fi	$b_1[x := v]$	H	
$x.i$	v_i	H	$H(x) = v$ or $\mathsf{mktag}(\tau, v)$ $1 \leq i \leq n, v = \langle v_1, \ldots, v_n\rangle$
$v_1\ v_2$	$b[f, x := v_1, v_2]$	H	$v_1 = \mathsf{fix}\, f(x:\tau_1):\tau_2.b$
let $x_1 = v_1$ and $x_2 = v_2$ in e'	$e'[x_1, x_2 := v_1, v_2]$	H	

Figure 4: Target Language Operational Semantics

and for invariance τ_t is σ. Using this new form, we can revise the type for z to $\mathsf{tag}^-(\alpha, \ldots)$, and type for *Failure* to $\mathsf{tag}^+(\mathsf{string}, \ldots)$. Then if z equals *Failure* we know that the types these tags were created to tag are equal. If σ is this type, we further know that $\alpha \leq \sigma$ since z has the contravariant tag type, and that $\sigma \leq \mathsf{string}$ as *Failure* has the covariant tag type. So $\alpha \leq \mathsf{string}$ and it safe to assume $x : \mathsf{string}$ in b_1.

The key is the relationship between the static tag type and the runtime tag type. In creating tagged values and subtags, we want the static tag type to be a subtype of the runtime tag type, and for destructing tagged values, we want the runtime tag type to be a subtype of the static type so that the type system conservatively refines types. The variance mechanism tracks and ensures the correct relationships.

Using these ideas, I present the target language in the next section, and then a translation from the tagging language to the target language in the following section.

4.1 Target Language

The target language incorporates the keys ideas of the previous section, that is, tag types with a variance mechanism and physical pointer equality, with a typical typed lambda calculus. The syntax appears in Figure 3. The operation $\mathsf{mktag}(\tau, \langle \vec{e}\rangle)$ creates a new tuple in the heap that can be used as a tag for the type τ; it has type $\mathsf{tag}^\circ(\tau, \langle \vec{\tau}\rangle)$ where $\vec{e} : \vec{\tau}$. The type $\mathsf{tag}^\phi(\tau_1, \tau_2)$ contains values of type τ_2 that are used as tags for the type τ_1. The value in this type may have been created as a tag for a subtype of τ_1 if ϕ is $+$, a supertype of τ_2 if ϕ is $-$, but only τ_1 if ϕ is $\circ$. Two values that tag types can be compared for physical pointer equality using the operation if $e_1 = e_2$ then b_1 else b_2 fi. This operation is asymmetric as it is intended to compare a tag for an unknown type, e_1, with a tag for a known type, e_2. If the two values are equal b_1 is executed, and e_2's tag type is used to refine e_1's; otherwise b_2 is executed. There are n-tuples and functions as before. In addition the translation will need recursive types, existentials, and option types. The recursive and existential types are standard. An option type $\tau?$ is either the value none^τ or the value $\mathsf{some}(v)$ for some $v : \tau$; the operation if? e_1 then $x.b_1$ else b_2 fi can be used to discriminate the two. I also include a special let form let $x_1 = e_1$ and $x_2 = e_2$ in e. This expression first evaluates e_1 to a value v_1, then evaluates e_2 to a value v_2, and the evaluates e with v_i substituted for x_i. It makes the proof of operational correctness simpler.

The operational semantics is similar in spirit to the tagging language, and is given in Figure 4. Like in the tagging language, the heap is used to remember identities, in particular, the identities of the tuples created. Two tuples, or pointers, are equal if they have the same address, that is, if they are the same variable. Hence the rules for the if construct.

The target language is eraseable. The operation $\mathsf{mktag}(\tau, e)$ is equivalent to e, and the annotation $\mathsf{mktag}(\tau, \cdot)$ on heap values is not needed at runtime.

The static semantics appears in Figures 5 and 6. Note that a typing context of the form $\Delta, \alpha \leq \tau$ must have $\mathrm{ftv}(\tau) \subseteq \Delta$. This syntactic restriction ensures that all typing contexts are well formed, which simplifies the proof of soundness. The static semantics is straightforward except for the tagging constructs. First, values in the type $\mathsf{tag}^\phi(\tau_t, \tau_{ds})$ also have type τ_{ds}, and the rule **tag-sub** reflects this specialised subsumption principle. This rule is used to manipulate the datastructure a type tag contains.

The two rules for tag comparison deserve mention.

$$\begin{array}{lrcl} \text{Typing Contexts} & \Delta & ::= & \epsilon \mid \Delta, \alpha \mid \Delta, \alpha \leq \tau \ (\text{ftv}(\tau) \subseteq \Delta) \\ \text{Value Contexts} & \Gamma & ::= & x_1 : \tau_1, \ldots, x_n : \tau_n \end{array}$$

$$\frac{}{\Delta \vdash_{\mathcal{T}} \tau} \ (\text{ftv}(\tau) \subseteq \Delta)$$

$$\frac{\Delta \vdash_{\mathcal{T}} \tau}{\Delta \vdash_{\mathcal{T}} \tau \leq \tau} \qquad \frac{\Delta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2 \quad \Delta \vdash_{\mathcal{T}} \tau_2 \leq \tau_3}{\Delta \vdash_{\mathcal{T}} \tau_1 \leq \tau_3} \qquad \frac{}{\Delta \vdash_{\mathcal{T}} \alpha \leq \tau} \ (\alpha \leq \tau \in \Delta)$$

$$\frac{\Delta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2 \quad \Delta \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_2}{\Delta \vdash_{\mathcal{T}} \mathsf{tag}^{\phi}(\tau_1, \sigma_1) \leq \mathsf{tag}^{+}(\tau_2, \sigma_2)} \ (\phi \in \{+, \circ\}) \qquad \frac{\Delta \vdash_{\mathcal{T}} \tau_2 \leq \tau_1 \quad \Delta \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_2}{\Delta \vdash_{\mathcal{T}} \mathsf{tag}^{\phi}(\tau_1, \sigma_1) \leq \mathsf{tag}^{-}(\tau_2, \sigma_2)} \ (\phi \in \{-, \circ\})$$

$$\frac{\Delta \vdash_{\mathcal{T}} \tau \quad \Delta \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_2}{\Delta \vdash_{\mathcal{T}} \mathsf{tag}^{\circ}(\tau, \sigma_1) \leq \mathsf{tag}^{\circ}(\tau, \sigma_2)}$$

$$\frac{\Delta \vdash_{\mathcal{T}} \mathsf{rec}\ \alpha.\tau_1 \quad \Delta \vdash_{\mathcal{T}} \mathsf{rec}\ \beta.\tau_2 \quad \Delta, \beta, \alpha \leq \beta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2}{\Delta \vdash_{\mathcal{T}} \mathsf{rec}\ \alpha.\tau_1 \leq \mathsf{rec}\ \beta.\tau_2} \ (\alpha \neq \beta; \alpha, \beta \notin \Delta)$$

$$\frac{\Delta, \alpha \vdash_{\mathcal{T}} \tau_1 \leq \tau_2}{\Delta \vdash_{\mathcal{T}} \exists \alpha.\tau_1 \leq \exists \alpha.\tau_2} \ (\alpha \notin \Delta) \qquad \frac{\Delta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2}{\Delta \vdash_{\mathcal{T}} \tau_1? \leq \tau_2?}$$

$$\frac{\Delta \vdash_{\mathcal{T}} \tau_i \leq \sigma_i}{\Delta \vdash_{\mathcal{T}} \langle \tau_1, \ldots, \tau_m \rangle \leq \langle \sigma_1, \ldots, \sigma_n \rangle} \ (m \geq n) \qquad \frac{\Delta \vdash_{\mathcal{T}} \tau_2 \leq \tau_1 \quad \Delta \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_2}{\Delta \vdash_{\mathcal{T}} \tau_1 \to \sigma_1 \leq \tau_2 \to \sigma_2}$$

Figure 5: Target Language Typing Rules for Types

Rule **t1** is for comparing an unknown tag against a known one. This is the rule used to type the translation, which always unpacks an existentially quantified package, extracts from it a tag for the quantified type, and compares it to a known tag. The rule requires the unknown tag e_1 to be a tag for a supertype of some type variable α, the known tag e_2 to be a tag for a subtype of a closed type. I assume, as in Java and ML, that classes and exception names are always associated with closed types. If the tags are equal then they must be tags for a type between the unknown type and the known type, that is, the unknown type is a subtype of the known one. The then branch b_1 is checked with this additional information.

However, rule **t1** is not closed under type substitution. In particular, if a closed type is substituted for α then the expression compares two tags for known types, and the rule no longer applies. Thus to prove type substitution and thus type soundness, the rule **t2** is used to type this case. It requires both e_1 and e_2 to be tags for known closed types σ_1 and σ_2 respectively. If $\epsilon \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_2$ does not hold then it is impossible for e_1 to be equal to e_2, therefore b_1 is only type checked when this condition holds. In fact, b_1 will probably not type check at all when this condition does not hold, as it may use values of type σ_1 where σ_2's are expected.

The static semantics is sound with respect to the operational semantics. The proof appears in the companion technical report [Gle99]. Standard techniques are used in the proof and the only difficulty is with the tag comparison operation. In showing type preservation for a successful tag comparison I use the fact that $\epsilon; \Gamma \vdash_{\mathcal{T}} x : \mathsf{tag}^{-}(\sigma_1, \tau_1)$ and $\epsilon; \Gamma \vdash_{\mathcal{T}} x : \mathsf{tag}^{+}(\sigma_2, \tau_2)$ implies $\epsilon \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_2$. Then by rule **t2** the then branch must type check. The other difference is the type substitution lemma mentioned earlier.

4.2 Translation

The translation from the tagging language to the target language is given in Figure 7. It is based on the ideas I sketched earlier. The key to the type translation is the translation of tag types. A tag for type τ is translated to a tuple with a tag option, suggesting the type $\mathsf{rec}\ \alpha.\mathsf{tag}^{-}([\![\tau]\!]_{\text{type}}, \langle \alpha? \rangle)$. This is not quite correct as the tag itself needs to be invariant, so the translation is actually one unrolling of this type with the outermost tag type invariant, $\mathsf{tag}^{\circ}([\![\tau]\!]_{\text{type}}, \langle(\mathsf{rec}\ \alpha.\mathsf{tag}^{-}([\![\tau]\!]_{\text{type}}, \langle \alpha? \rangle))?\rangle)$, except the option type is shifted into the recursive type, $\mathsf{tag}^{\circ}([\![\tau]\!]_{\text{type}}, \langle \mathsf{rec}\ \alpha.\mathsf{tag}^{-}([\![\tau]\!]_{\text{type}}, \langle \alpha \rangle)?\rangle)$. The translation of tagged is an existential abstracting over α a pair of a tag for α and an α. The operations $\mathsf{newtag}(\tau)$, $\mathsf{subtag}(\tau, e)$, and $\mathsf{mktagged}(e)$ are translated as I described earlier modulo all the typing annotations needed for recursive types, option types, and existential types. The $tagchk^{H}(x, y)$ predicate is reified as a recursive function, $tagchk(y_1, y_2, \tau, \sigma)$, that searches the superchain and then returns a σ option where σ is the known type. The translation of the tag comparison operation uses the let form to evaluate the arguments, then unpacks the tagged value, uses the reified tag check predicate to do the comparison, and then executes the appropriate translated branch.

Technically the translation is type directed as it needs type information in two places. Thus, the translation may not be defined for all source terms, but it is easy to show that it is defined for all typeable source terms. Furthermore, because the tag type is invariant it is easy to show that there is only one type possible in the places where type information is required, so the translation is coherent. Rather than presenting the translation as a acting on typing derivations, I have indicated the necessary type information with a $: \tau$ notation on the source terms.

The translation is both type preserving and operationally correct. These theorems are stated below and are proven in the companion technical report [Gle99]. Proving type preservation is a straightforward inductive proof. Because I carefully chose the tagging and target language semantics to match on common constructs, and through the use of the let construct, operational correctness can be proven by a

$$\frac{\Delta;\Gamma \vdash_{\mathcal{T}} e : \tau_1 \quad \Delta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2}{\Delta;\Gamma \vdash_{\mathcal{T}} e : \tau_2} \qquad [\textbf{rule tag-sub}]\ \frac{\Delta;\Gamma \vdash_{\mathcal{T}} e : \mathsf{tag}^{\phi}(\tau,\sigma)}{\Delta;\Gamma \vdash_{\mathcal{T}} e : \sigma}$$

$$\frac{}{\Delta;\Gamma \vdash_{\mathcal{T}} x : \tau}\ (\Gamma(x) = \tau) \qquad \frac{\Delta;\Gamma \vdash_{\mathcal{T}} e_i : \tau_i \quad \Delta \vdash_{\mathcal{T}} \tau}{\Delta;\Gamma \vdash_{\mathcal{T}} \mathsf{mktag}(\tau, \langle e_1, \ldots, e_n\rangle) : \mathsf{tag}^{\circ}(\tau, \langle \tau_1, \ldots, \tau_n\rangle)}$$

$$[\textbf{rule t1}]\ \frac{\begin{array}{c}\epsilon \vdash_{\mathcal{T}} \sigma \\ \Delta;\Gamma \vdash_{\mathcal{T}} e_1 : \mathsf{tag}^{-}(\alpha,\tau_1) \\ \Delta;\Gamma \vdash_{\mathcal{T}} e_2 : \mathsf{tag}^{+}(\sigma,\tau_2) \\ \Delta_1, \alpha \leq \sigma, \Delta_2;\Gamma \vdash_{\mathcal{T}} b_1 : \tau \\ \Delta;\Gamma \vdash_{\mathcal{T}} b_2 : \tau\end{array}}{\Delta;\Gamma \vdash_{\mathcal{T}} \mathsf{if}\ e_1 = e_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} : \tau}\ (\Delta = \Delta_1, \alpha, \Delta_2)$$

$$[\textbf{rule t2}]\ \frac{\begin{array}{c}\Delta;\Gamma \vdash_{\mathcal{T}} e_1 : \mathsf{tag}^{-}(\sigma_1,\tau_1) \quad \epsilon \vdash_{\mathcal{T}} \sigma_1 \\ \Delta;\Gamma \vdash_{\mathcal{T}} e_2 : \mathsf{tag}^{+}(\sigma_2,\tau_2) \quad \epsilon \vdash_{\mathcal{T}} \sigma_2 \\ (\epsilon \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_2 \quad \Rightarrow \quad \Delta;\Gamma \vdash_{\mathcal{T}} b_1 : \tau) \\ \Delta;\Gamma \vdash_{\mathcal{T}} b_2 : \tau\end{array}}{\Delta;\Gamma \vdash_{\mathcal{T}} \mathsf{if}\ e_1 = e_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} : \tau}$$

$$\frac{\Delta;\Gamma \vdash_{\mathcal{T}} e : \sigma[\alpha := \tau]}{\Delta;\Gamma \vdash_{\mathcal{T}} \mathsf{roll}^{\tau}(e) : \tau}\ (\tau = \mathsf{rec}\ \alpha.\sigma) \qquad \frac{\Delta;\Gamma \vdash_{\mathcal{T}} e : \tau}{\Delta;\Gamma \vdash_{\mathcal{T}} \mathsf{unroll}(e) : \sigma[\alpha := \tau]}\ (\tau = \mathsf{rec}\ \alpha.\sigma)$$

$$\frac{\Delta \vdash_{\mathcal{T}} \tau_1 \quad \Delta;\Gamma \vdash_{\mathcal{T}} e : \tau_2[\alpha := \tau_1]}{\Delta;\Gamma \vdash_{\mathcal{T}} \mathsf{pack}[\tau_1, e]\ \mathsf{as}\ \exists\alpha.\tau_2 : \exists\alpha.\tau_2}$$

$$\frac{\Delta;\Gamma \vdash_{\mathcal{T}} e_1 : \exists\alpha.\tau_1 \quad \Delta, \alpha;\Gamma, x : \tau_1 \vdash_{\mathcal{T}} e_2 : \tau \quad \Delta \vdash_{\mathcal{T}} \tau}{\Delta;\Gamma \vdash_{\mathcal{T}} \mathsf{unpack}[\alpha, x] = e_1\ \mathsf{in}\ e_2 : \tau}\ (\alpha \notin \Delta, x \notin \Gamma)$$

$$\frac{\Delta \vdash_{\mathcal{T}} \tau}{\Delta;\Gamma \vdash_{\mathcal{T}} \mathsf{none}^{\tau} : \tau?} \qquad \frac{\Delta;\Gamma \vdash_{\mathcal{T}} e : \tau}{\Delta;\Gamma \vdash_{\mathcal{T}} \mathsf{some}(e) : \tau?}$$

$$\frac{\Delta;\Gamma \vdash_{\mathcal{T}} e_1 : \sigma? \quad \Delta;\Gamma, x : \sigma \vdash_{\mathcal{T}} b_1 : \tau \quad \Delta;\Gamma \vdash_{\mathcal{T}} b_2 : \tau}{\Delta;\Gamma \vdash_{\mathcal{T}} \mathsf{if?}\ e_1\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi} : \tau}\ (x \notin \Gamma)$$

$$\frac{\Delta;\Gamma \vdash_{\mathcal{T}} e_i : \tau_i}{\Delta;\Gamma \vdash_{\mathcal{T}} \langle e_1, \ldots, e_n\rangle : \langle \tau_1, \ldots, \tau_n\rangle} \qquad \frac{\Delta;\Gamma \vdash_{\mathcal{T}} e : \langle \tau_1, \ldots, \tau_n\rangle}{\Delta;\Gamma \vdash_{\mathcal{T}} e.i : \tau_i}\ (1 \leq i \leq n)$$

$$\frac{\Delta \vdash_{\mathcal{T}} \tau_1 \to \tau_2 \quad \Delta;\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash_{\mathcal{T}} b : \tau_2}{\Delta;\Gamma \vdash_{\mathcal{T}} \mathsf{fix}\, f(x : \tau_1) : \tau_2.b : \tau_1 \to \tau_2}\ (f, x \notin \Gamma)$$

$$\frac{\Delta;\Gamma \vdash_{\mathcal{T}} e_1 : \tau_1 \to \tau_2 \quad \Delta;\Gamma \vdash_{\mathcal{T}} e_2 : \tau_1}{\Delta;\Gamma \vdash_{\mathcal{T}} e_1\ e_2 : \tau_2}$$

$$\frac{\Delta;\Gamma \vdash_{\mathcal{T}} e_1 : \tau_1 \quad \Delta;\Gamma \vdash_{\mathcal{T}} e_2 : \tau_2 \quad \Delta;\Gamma, x_1{:}\tau_1, x_2{:}\tau_2 \vdash_{\mathcal{T}} e : \tau}{\Delta;\Gamma \vdash_{\mathcal{T}} \mathsf{let}\ x_1 = e_1\ \mathsf{and}\ x_2 = e_2\ \mathsf{in}\ e : \tau}\ (x_1, x_2 \notin \Gamma)$$

$$\frac{\epsilon;\epsilon \vdash_{\mathcal{T}} h_1 : \tau_1 \quad \ldots \quad \epsilon; x_1 : \tau_1, \ldots, x_{n-1} : \tau_{n-1} \vdash_{\mathcal{T}} h_n : \tau_n}{\vdash_{\mathcal{T}} x_1 = h_1, \ldots, x_n = h_n : x_1 : \tau_1, \ldots, x_n : \tau_n}\ (x_1, \ldots, x_n \text{ distinct})$$

$$\frac{\vdash_{\mathcal{T}} H : \Gamma \quad \epsilon;\Gamma \vdash_{\mathcal{T}} e : \tau}{\vdash_{\mathcal{T}} \mathsf{let}\ H\ \mathsf{in}\ e : \tau}$$

Figure 6: Target Language Typing Rules for Terms

$$
\begin{array}{lcl}
tag(\phi,\tau) & = & \mathsf{tag}^{\phi}(\tau,\langle tag'(\tau)\rangle)\\
tag'(\tau) & = & \mathsf{rec}\ \alpha.\mathsf{tag}^{-}(\tau,\langle\alpha\rangle)?\\
[\![\mathsf{tag}(\tau)]\!]_{\mathrm{type}} & = & tag(\circ,[\![\tau]\!]_{\mathrm{type}})\\
[\![\mathsf{tagged}]\!]_{\mathrm{type}} & = & \exists\alpha.\langle tag(-,\alpha),\alpha\rangle\\
[\![\langle\tau_1,\ldots,\tau_n\rangle]\!]_{\mathrm{type}} & = & \langle[\![\tau_1]\!]_{\mathrm{type}},\ldots,[\![\tau_n]\!]_{\mathrm{type}}\rangle\\
[\![\tau_1\to\tau_2]\!]_{\mathrm{type}} & = & [\![\tau_1]\!]_{\mathrm{type}}\to[\![\tau_2]\!]_{\mathrm{type}}\\
[\![x_1{:}\tau_1,\ldots,x_n{:}\tau_n]\!]_{\mathrm{ctxt}} & = & x_1{:}[\![\tau_1]\!]_{\mathrm{type}},\ldots,x_n{:}[\![\tau_n]\!]_{\mathrm{type}}\\
\\
[\![x]\!]_{\mathrm{exp}} & = & x\\
[\![\mathsf{newtag}(\tau)]\!]_{\mathrm{exp}} & = & \mathsf{mktag}([\![\tau]\!]_{\mathrm{type}},\langle\mathsf{roll}^{tag'([\![\tau]\!]_{\mathrm{type}})}(\mathsf{none}^{tag(-,[\![\tau]\!]_{\mathrm{type}})})\rangle)\\
[\![\mathsf{subtag}(\tau,e)]\!]_{\mathrm{exp}} & = & \mathsf{mktag}([\![\tau]\!]_{\mathrm{type}},\langle\mathsf{roll}^{tag'([\![\tau]\!]_{\mathrm{type}})}(\mathsf{some}([\![e]\!]_{\mathrm{exp}}))\rangle)\\
[\![\mathsf{mktagged}(e:\langle\mathsf{tag}(\tau),\tau\rangle)]\!]_{\mathrm{exp}} & = & \mathsf{pack}[[\![\tau]\!]_{\mathrm{type}},[\![e]\!]_{\mathrm{exp}}]\ \mathsf{as}\ [\![\mathsf{tagged}]\!]_{\mathrm{type}}\\
tagchk(y_1,y_2,\tau,\sigma) & = & \mathsf{fix}\,y_3(y_4:tag(-,\tau)):\sigma?.\\
 & & \quad\mathsf{if}\ \ y_4=y_2\ \mathsf{then}\ \mathsf{some}(y_1.2)\ \mathsf{else}\\
 & & \quad\mathsf{if?}\ \mathsf{unroll}(y_4.1)\ \mathsf{then}\ y_5.y_3\ y_5\ \mathsf{else}\ \mathsf{none}^{\sigma}\ \mathsf{fi}\ \ \mathsf{fi}\\
\\
[\![\mathsf{if}\ \mathsf{tagof}(e_1)\le e_2:\mathsf{tag}(\sigma)\ & & \\
\mathsf{then}\ x.b_1\ \ \mathsf{else}\ \ b_2\ \ \mathsf{fi}]\!]_{\mathrm{exp}} & = & \mathsf{let}\ x_1=[\![e_1]\!]_{\mathrm{exp}}\ \mathsf{and}\ x_2=[\![e_2]\!]_{\mathrm{exp}}\ \mathsf{in}\ \mathsf{unpack}[\alpha,y_1]=x_1\ \mathsf{in}\\
 & & \mathsf{if?}\ tagchk(y_1,x_2,\alpha,[\![\sigma]\!]_{\mathrm{type}})\ y_1.1\ \mathsf{then}\ x.[\![b_1]\!]_{\mathrm{exp}}\ \mathsf{else}\ [\![b_2]\!]_{\mathrm{exp}}\ \mathsf{fi}\\
[\![\langle e_1,\ldots,e_n\rangle]\!]_{\mathrm{exp}} & = & \langle[\![e_1]\!]_{\mathrm{exp}},\ldots,[\![e_n]\!]_{\mathrm{exp}}\rangle\\
[\![e.i]\!]_{\mathrm{exp}} & = & [\![e]\!]_{\mathrm{exp}}.i\\
[\![\mathsf{fix}\,f(x{:}\tau_1){:}\tau_2.e]\!]_{\mathrm{exp}} & = & \mathsf{fix}\,f(x{:}[\![\tau_1]\!]_{\mathrm{type}}){:}[\![\tau_2]\!]_{\mathrm{type}}.[\![e]\!]_{\mathrm{exp}}\\
[\![e_1\ e_2]\!]_{\mathrm{exp}} & = & [\![e_1]\!]_{\mathrm{exp}}\ [\![e_2]\!]_{\mathrm{exp}}\\
\\
[\![(\tau,\epsilon)]\!]_{\mathrm{hval}} & = & \mathsf{mktag}([\![\tau]\!]_{\mathrm{type}},\langle\mathsf{roll}^{tag'([\![\tau]\!]_{\mathrm{type}})}(\mathsf{none}^{tag(-,[\![\tau]\!]_{\mathrm{type}})})\rangle)\\
[\![(\tau,x)]\!]_{\mathrm{hval}} & = & \mathsf{mktag}([\![\tau]\!]_{\mathrm{type}},\langle\mathsf{roll}^{tag'([\![\tau]\!]_{\mathrm{type}})}(\mathsf{some}(x))\rangle)\\
[\![\langle v_1,\ldots,v_n\rangle]\!]_{\mathrm{hval}} & = & \langle[\![v_1]\!]_{\mathrm{exp}},\ldots,[\![v_n]\!]_{\mathrm{exp}}\rangle\\
[\![x_1=h_1,\ldots,x_n=h_n]\!]_{\mathrm{heap}} & = & x_1=[\![h_1]\!]_{\mathrm{hval}},\ldots,x_n=[\![h_n]\!]_{\mathrm{hval}}\\
[\![\mathsf{let}\ H\ \mathsf{in}\ e]\!]_{\mathrm{prog}} & = & \mathsf{let}\ [\![H]\!]_{\mathrm{heap}}\ \mathsf{in}\ [\![e]\!]_{\mathrm{exp}}
\end{array}
$$

Figure 7: The Translation

simulation argument. Most of the work is in showing that tag checks are implemented properly.

Theorem 4.1 *If* $\vdash_{\mathcal{S}} P:\tau$ *then* $\vdash_{\mathcal{T}} [\![P]\!]_{\mathrm{prog}}:[\![\tau]\!]_{\mathrm{type}}$.

Theorem 4.2 *If* $\vdash_{\mathcal{S}} P_1:\tau$ *and* $P_1\mapsto P_2$ *then* $[\![P_1]\!]_{\mathrm{prog}}\mapsto^{+}[\![P_2]\!]_{\mathrm{prog}}$. *(An appropriate converse also holds.)*

5 Extensions and Implementation

The implementation given is quite naive, although it requires only constant space for each tag, it takes time linear in the height of the tag hierarchy for each tag check. Java implementations typically trade space for time, and represent a tag as an array of all of the tag's ancestors. This implementation requires only one array subscript and one physical pointer equality test per tag check, but uses space proportional to the height of the tag hierarchy per tag. Also, this scheme is not suitable for multiple inheritance hierarchies. The companion technical report [Gle99] shows how to express this array scheme in a variant of the target language. I expect that most schemes for implementing multiple inheritance hierarchies that use pointer comparisons, could also be expressed in variants of the target language.

For expository purposes, I used a type lambda calculus as the target of the translation. However, the ideas could be applied to other languages. In fact, they were used to augment TALx86 [MCG+99], a typed assembly language [MWCG98] for Intel's IA32 architecture, to provide support for exceptions and downcasting in object oriented languages. This implementation was straightforward: One type constructor was extended, and two instructions and the static data mechanism were changed to use this extended type constructor. The TALx86 implementation includes a compiler for a language called Popcorn, a safe C-like language. Popcorn has ML style exceptions, and the compiler was modified to use the new type tagging mechanisms instead of the old ad hoc mechanisms. Furthermore, the array scheme mentioned above could be implemented in the augmented TALx86.

6 Related Work

Reppy and Riecke [RR96] describe an extension of SML with objects. Their language contains a class case mechanism, and they discuss the connections of this mechanism with hierarchy extensible sums and exceptions. Their paper includes a formalised core calculus, and a sketch of their implementation. There are three important differences between their system and the one presented in this paper. First, their tags are second class, whereas mine are first class. Second, their type system is named, whereas mine is structural. Third, they do not formalise their implementation nor discuss the typing issues that arise. Harper and Stone [HS97] formalise a non-hierarchical extensible sum for modelling exceptions in a type theoretic account of SML. Their extensible sum is like my tagging language without subtags. However, they do not discuss implementation issues.

Type dispatch in a structurally typed language was first discussed by Abadi *et al.* [ACPP91]. Harper and Mor-

risett [HM95, Mor95] discuss the related problem of type analysis, and formalise a type dispatch language for structural types without subtyping. Crary, Weirich, and Morrisett's work on λ^R and LX [CWM98, CW99] shows how to do type analysis in a type erasure framework. Their work is the only work I know of that addresses the implementation of type dispatch for structural types. I know of no work that addresses structural types with subtyping.

I think it is worth spelling out the two key differences between the type analysis of structural types and my problem. First, my tagging language is concerned with matching whole types and their subtypes, whereas λ^R is concerned with matching the top level structure of types: is v in τ versus does v have an arrow type and what are tags for the argument and result type. To achieve the type refinement of my target language a λ^R implementation would have to crawl over the entire structure of a tag for the unknown type, whereas my language can make the refinement with one pointer equality test. Second, my language is concerned with branded types whereas λ^R is concerned with the types themselves. Consider my informal coding of exceptions. In O'Caml there are at least three exceptions that carry strings. Therefore, there will be at least three tags for the type string, and the distinction between a string tagged with `Failure` and the same string tagged with `Invalid_argument` is important. In the λ^R setting this distinction is irrelevant, only the fact that the value is a string is important.

7 Summary

This paper has shown that a number of language mechanisms, among them class casting, class casing, exceptions, and extensible hierarchical sums, share a common mechanism: type tagging. A tagging language containing this core mechanism was defined and then translated into a more primitive language with just the notion of values being tags for types and physical pointer equality. The type soundness of the target language and the type preservation and operational correctness of the translation was proven. Thus, this paper provides a solid theoretical foundation for all the mechanisms mentioned above. Along with the work in type analysis of structural types, this work provides foundations for a theory of type dispatch in programming languages.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory Of Objects*. Springer-Verlag, 1996.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Progamming Languages and Systems*, 13(2):237–268, April 1991.

[CGL95] Giuseppe Castanga, Giorgio Ghelli, and Guiseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[Cha97] Craig Chambers. The Cecil language, specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, Washington 98195-2350, USA, March 1997.

[CW99] Karl Crary and Stephanie Weirich. Flexible type analysis. In *1999 ACM SIGPLAN International Conference on Functional Programming*, Paris, France, September 1999. This volume.

[CWM98] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *1998 ACM SIGPLAN International Conference on Functional Programming*, pages 301–312, Baltimore Maryland, USA, September 1998.

[FR99] Kathleen Fisher and John Reppy. The design of a class mechanism for Moby. In *1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, USA, May 1999. Moby information is available at `http://www.cs.bell-labs.com/~jhr/moby`.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Gle99] Neal Glew. Type dispatch for named hierarchical types. Technical Report TR99-1738, Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA, April 1999. Available at `http://www.cs.cornell.edu/glew/paper-list.html`.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, USA, January 1995.

[HS97] Robert Harper and Christopher Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, Carnegie Mellon University, Pittsburgh, PA 15213, June 1997.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998. With H. Abelson, N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, and M. Wand.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, Daivd Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, volume 0228 of *INRIA Research Reports*, Atlanta, GA, USA, May 1999.

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *ACM Conference on Functional Programming and Computer Architecture*, 1995.

[Mor95] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, December 1995. Published as CMU Technical Report CMU-CS-95-226.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego California, USA, January 1998. ACM Press.

[RR96] John Reppy and Jon Riecke. Simple objects for Standard ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–180. ACM Press, 1996.

[TMC+96] David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, USA, May 1996.