Type Dispatch for Named Hierarchical Types *

Neal Glew
Cornell University

7 April 1999

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 dispatch branches. Existing work in functional languages has addressed certain kinds of type dispatch, namely, intensional type analysis. However, this work does not extend to languages with subtyping nor to named types.

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 formalise a small tagging language. After discussing how to implement the tagging language, I present a more primitive language and give a formal translation from the tagging language.

1 Introduction

A number of programming languages allow dispatch on the runtime type of a value. For example, in Scheme [KCR98] there are predicates for testing if a value is the empty list, an integer, a pair, and so on. In many object oriented languages there are constructs for testing or switching on the class of an object.[1] For example, Java [GJS96] has a cast expression and an exception catch mechanism for switching on the class of the exception object. The cast expression `(`*classname*`)`e tests whether e's runtime class is in the hierarchy under the given class; if so, it refines e's type to the given class; if not, an exception is thrown. 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 against *classname*$_1$ through *classname*$_n$. If *classname*$_i$ is the first matching class, x_i is bound to the exception with type *classname*$_i$, and *blk*$_i$ is executed. A final example of type dispatch is λ_i^{ML} [HM95, Mor95], the basis for an intermediate language in the TIL compiler [TMC+96]. The λ_i^{ML} calculus has a typecase construct for determining the outermost type constructor of an unknown type. The typecase

*This material 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.

[1] Classes are closely related to runtime types; I will often use a class when I mean the corresponding type.

construct is used to implement specialised data representations in the presence of polymorphism and for writing polymorphic print and marshaling functions.

Scheme's type predicates make the language much more expressive. Java's implementation of exceptions crucially relies upon a class matching mechanism. The TIL compiler's use of runtime type examination leads to significant optimisation benefits. The importance of type dispatch constructs justifies seeking formal foundations for them.

In typed languages, type refinement is a key property of type dispatch constructs: After switching on the runtime type of a value, that value's static type changes to reflect the type being compared against. Consider the Java code (note that `FileNotFound` is a subclass of `IOException`, which is a descendant of `Throwable`):

`try { throw new FileNotFound(); } catch (IOException x)` *blk*`;`

The try body throws an exception that has runtime type `FileNotFound`, but Java's type rule for try gives the exception static type `Throwable`. The exception matches against the catch clause, and `x` is bound to the exception with static type `IOException`, a refinement of `Throwable`. A similar sort of refinement occurs in λ_i^{ML}. For example, consider:

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

x initially has static type α, but in expression e_1 it is refined to $\alpha_1 \times \alpha_2$.

While type dispatch constructs in functional languages have been studied formally [HM95, Mor95, CWM98, CW99, *etc.*], this work does not extend to object oriented languages as it neglects two important features: subtypes and the creation of "new" (*i.e.* generative) types. To my knowledge, this paper is the first to formalise type dispatch constructs common in object oriented languages. The tagging language I will define handles both features mentioned above, and is quite general: It explains class cast and class case constructs in object oriented languages, ML style exceptions, and hierarchical extensible sums, and could be used in a typed compilation of multidispatch languages such as Cecil [Cha97].

Languages with type dispatch pose another problem. Consider compiling type dispatch to Typed Assembly Language [MWCG98, MCG$^+$99]. Since TAL does not have type dispatch primitives, the compiler writer has to design data structures and algorithms to use these data structures to perform type dispatch. Furthermore, these data structures and algorithms have to type check. TAL's current type system is not expressive enough to type check a typical implementation. To address this inexpressiveness, this paper defines a typed target language without type dispatch primitives and a formal translation. Correctness theorems for the translation are stated and proven in a companion technical report [MCG$^+$99]. I used the key ideas in the target language to extend TAL [MCG$^+$99] so that the type dispatch mechanism described in this paper can be compiled to TAL; 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 mentioned earlier and discuss its implementation. I begin by describing in more detail the programming language constructs being addressed. From these constructs I extract a common theme, 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 at the type implied by 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 be 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 exception packets are an example of extensible sums. The exception type is like a giant 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. Hierarchical extensible sum types are being considered for ML2000, and appear 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

```
let Failure = newtag(string) in                       // exception Failure of string
let ep1 = tagged(⟨Failure,"unimplemented"⟩) in        // ep1 = Failure "unimplemented"
⋮
iftagof ep1 = Failure then                            // match ep1 with
   x.printf "Computation failed: %s" x                //    Failure(x) -> ···
else
   printf "Some other exception"                      // |  _ -> ···
fi
```

Figure 1: Exceptions Coded in the Tagging Language

> 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.

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* or a *tag case* construct with type refinement in the successful branches of the test(s). 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. Often, the tests are not "is tag t_1 equal to tag t_2" but "is tag t_1 in the subhierarchy under tag t_2". 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: tagging a value with a type tag, comparing the tag of a tagged value against known tags, and creating hierarchies of type tags. A new tag is created by one of two operations: $\mathsf{newtag}(\tau)$ or $\mathsf{subtag}(e, \tau)$. 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 $\mathsf{subtag}(e, \cdot)$ form creates a subtag of tag e. Values are tagged with the operation $\mathsf{tagged}(\langle e_1, e_2\rangle)$ where e_1 is the type tag, and e_2 the value to be tagged. The result is a value of type tagged. Tagged values are compared against known tags with the operation $\mathsf{iftagof}\, e_1 = e_2\, \mathsf{then}\, x.b_1\, \mathsf{else}\, b_2\, \mathsf{fi}$ 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.

An example of how to use these operations to code exceptions appears in Figure 1. For expository purposes, the example uses some constructs not in the formal language.

The tagging language has the above operations plus n-tuples and functions with their usual

Syntax:

Values	v	$::=$	$x \mid \mathsf{tagged}(v) \mid \mathsf{fix}\, f(x{:}\tau_1){:}\tau_2.b$
Contexts	E	$::=$	$[] \mid \mathsf{subtag}(E,\tau) \mid \mathsf{tagged}(E) \mid \langle \vec{v}, E, \vec{e}\rangle \mid E.i \mid E\ e \mid v\ E \mid$ $\mathsf{iftagof}\ E = e_2\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid \mathsf{iftagof}\ v = E\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$
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
$\mathsf{subtag}(y,\tau)$	x	$H, x = (\tau, y)$	x fresh
$\langle \vec{v}\rangle$	x	$H, x = I$	x fresh
$x.i$	v_i	H	$H(x) = \langle v_1, \ldots, v_n\rangle; 1 \le i \le n$
$v_1\ v_2$	$b[f, x := v_1, v_2]$	H	$v_1 = \mathsf{fix}\, f(x{:}\tau_1){:}\tau_2.b$
$\mathsf{iftagof}\ \mathsf{tagged}(x) = y$ $\mathsf{then}\ z.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$	$b_1[z := v]$	H	$H(x) = \langle x', v, \vec{v}\rangle; \mathit{tagchk}^H(x', y)$
$\mathsf{iftagof}\ \mathsf{tagged}(x) = y$ $\mathsf{then}\ z.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$	b_2	H	$H(x) = \langle x', v, \vec{v}\rangle$; not $\mathit{tagchk}^H(x', y)$

Tag checking:

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

Figure 2: Source Language Operational Semantics

introduction and elimination forms. The syntax is:

Types	τ, σ	$::=$	$\mathsf{tag}(\tau) \mid \mathsf{tagged} \mid \langle \vec{\tau}\rangle \mid \tau_1 \to \tau_2$
Terms	e, b	$::=$	$x \mid \mathsf{newtag}(\tau) \mid \mathsf{subtag}(e,\tau) \mid \mathsf{tagged}(e) \mid \mathsf{iftagof}\ e_1 = e_2\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid$ $\langle \vec{e}\rangle \mid e.i \mid \mathsf{fix}\, f(x : \tau_1) : \tau_2.b \mid e_1\ e_2$

The notation $\vec{X}$ denotes a vector of objects drawn from syntactic category X. The f and x in $\mathsf{fix}\, f(x{:}\tau_1){:}\tau_2.b$ bind in b and the x in $\mathsf{iftagof}\ e_1 = e_2\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$ binds in b_1. I consider syntactic objects equal up to α-equivalence.

The operational semantics is given in Figure 2. The generation of new tags is modeled by an allocation style semantics (*c.f.* Morrisett *et al.* [MFH95]). In an allocation style semantics terms are evaluated in the presence of a heap, H. Heaps are intended to model the store, and they map variables to heap values h. Variables are like pointers into memory, and heap values are like the contents of the memory pointed to. An advantage of this model is that it captures identity: Different tags get different variables even if they tag the same type and have the same supertag. For the tagging language, heap values are either tuples of values, $\langle \vec{v}\rangle$, or tag definitions (τ, s) where τ is the type being tagged and s is the optional supertag (ϵ denotes the absence of a supertag).

The most interesting rule is the one for $\mathsf{iftagof} \cdot = \cdot \mathsf{then}\ \cdot\ \mathsf{else}\ \cdot\ \mathsf{fi}$. The comparison of a tag and its ancestors against a known tag is formalised by the predicate $\mathit{tagchk}^H(x, y)$ where H contains the tag definitions, x the unknown tag, and y the known tag. The definition of this predicate essentially says that either x and y are the same tag or x has a supertag and the predicate holds for 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 in the depth of x in the tag hierarchy. 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 a context and substitution based reduction semantics. Note that the semantics is deterministic, call by value, and left to right.

The typing rules appear in Figure 3, and consist of judgements for subtyping and expression, heap value, heap, and program state typing. Subtyping for tag and tagged types is trivial, and the subtyping for tuples and functions is standard. Note that reflexivity and transitivity of subtyping is derivable from the rules given. The rule for subtag requires that e be a tag for type τ' and that τ be a subtype of τ', the later ensures that types associated with tags form a subtype hierarchy in parallel to the tag hierarchy. The rule for $\mathsf{tagged}(e)$ requires that e be a pair of a type tag for τ and a value of type τ. The rule for $\mathsf{iftagof}\ e_1 = e_2\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$ requires e_1 to have type tagged, e_2 to be a tag for 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 $\mathit{tagchk}^H(x, y)$ then $\vdash_{\mathcal{S}} \sigma_1 \leq \sigma_2$.*

4 Implementation

Real machines do not provide primitives like $\mathsf{newtag}(\tau)$, $\mathsf{subtag}(e, \tau)$, $\mathsf{tagged}(e)$, and $\mathsf{iftagof}\ e_1 = e_2\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$. Compiler writers must select data structures to represent the tags and algorithms to implement tag comparisons. 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 example in Figure 1 ignoring types. To create the new tag *Failure* the compiler would allocate a new heap block storing in it the optional

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3: Source Language Typing Rules

supertag. To create the tagged value $ep1$ the compiler would create a pair consisting of $Failure$ and the literal string. So the first two lines might become:

let $Failure = \langle \mathsf{none} \rangle$ in
let $ep1 = \langle Failure, \text{“unimplemented”} \rangle$ in

To translate the last part of the example, the compiler would extract the tag in $ep1$, which is just a heap pointer, and compare it against $Failure$. If they are equal x would be bound to the second component of $ep1$ 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:

let $z = ep1.1$ in
$loop$: if $z = Failure$ then let $x = ep1.2$ in b_1
    else match $z.1$ with none $\to b_2$ | some$(z') \to z := z'$; goto $loop$

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 $ep1$ is unknown, yet if the comparison $z = Failure$ succeeds, the type of $ep1.2$ 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 $ep1$. 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 $ep1$ together, so that type information generated by the comparison will change $ep1$'s type.

The solution is to introduce a new type for the values being used to implement type tags, and to use type variables to link tags to values being tagged. The target type $\mathsf{tag}(\tau_1, \tau_2)$ is similar to the tagging language tag type. However, where the tagging language hides the structure of the tag, the target language allows the user to specify the structure through the additional argument τ_2. In particular, a value v is in the type $\mathsf{tag}(\tau_1, \tau_2)$ if v also has type τ_2 and the programmer declared v to be a tag for type τ_1. Continuing the example, $Failure$ is essentially a linked list with no data, which has type $\mathsf{rec}\ \alpha.\langle\alpha?\rangle$ (where $\tau?$ is an option type), except that the nodes are also tags for string. So $Failure$ has type $\mathsf{rec}\ \alpha.\mathsf{tag}(\mathsf{string}, \langle\alpha?\rangle)$. The tagged value $ep1$ is a pair of such a tag and a value except that the type of the value is abstracted, thus $ep1$ has type $\exists\beta.\langle\mathsf{rec}\ \alpha.\mathsf{tag}(\beta, \langle\alpha?\rangle), \beta\rangle$. To get the initial value of z, $ep1$ is unpacked introducing β into the type context. Thus z has type $\mathsf{rec}\ \alpha.\mathsf{tag}(\beta, \langle\alpha?\rangle)$. If $z = Failure$ succeeds then the type z tags and the type $Failure$ tags must be the same, that is, β is string. The target type system will use this in type checking $\mathsf{let}\ x = ep1.2\ \mathsf{in}\ b_1$. Since $ep1.2$ has type β, which is string, x will get type the correct type.

Two complications arise with this basic scheme. The first concerns how to create values of type $\mathsf{tag}(\tau_1, \tau_2)$. Ideally, target code would just declare that a value of type τ_2 was being used as a tag for τ_1 with an operation $\mathsf{tag}(e, \tau)$. However, this is unsound as the same value may be declared a tag for several types. To see this unsoundness consider the following malicious code:

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

The variables x_1, x_2, and x_3 are all bound to the same heap value, 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:

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

The body of *foo* compares $z.1$ a tag for β to x_3 a tag for float. Thus 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 of 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 $\mathsf{tag}(e, \tau)$ 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 $\mathsf{tag}(e, \tau)$ only at points where new heap values are created. For example, $\langle\vec{e}\rangle$ creates a new heap value; the target operation $\mathsf{tag}(\langle\vec{e}\rangle, \tau)$ 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. Consider a subtag of $Failure$, $MyFailure$, that tags type $\mathsf{string}[10]$ (the type for strings of length 10).

(Note that $\mathsf{string}[10]$ is a subtype of string.) The translated code for $MyFailure$ is:

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

Consider how this type checks. As discussed before $Failure$ has type $\mathsf{rec}\ \alpha.\mathsf{tag}(\mathsf{string}, \langle \alpha?\rangle)$, and $MyFailure$ should have type $\mathsf{rec}\ \alpha.\mathsf{tag}(\mathsf{string}[10], \langle \alpha?\rangle)$. However, the right hand side above has type $\mathsf{tag}(\mathsf{string}[10], \langle (\mathsf{rec}\ \alpha.\mathsf{tag}(\mathsf{string}, \langle \alpha?\rangle))?\rangle)$. The later type would be an unrolling of the desired type if $\mathsf{tag}(\mathsf{string}, \ldots)$ was a subtype of $\mathsf{tag}(\mathsf{string}[10], \ldots)$. Thus it seems that in creating the data structures that implement type tags the tag type needs to be contravariant in its first argument.

However, making the tag type contravariant in the first argument is unsound for tag comparisons. Consider again, the code from above:

$$\mathsf{if}\ z.1 = Failure\ \mathsf{then}\ \mathsf{let}\ x = ep1.2\ \mathsf{in}\ b_1\ \mathsf{else}\ \ldots\ \mathsf{fi}$$

Under the contravariant rule $Failure$ subsumes to a tag type for $\mathsf{string}[10]$, so the type system could type check $\mathsf{let}\ x = ep1.2\ \mathsf{in}\ b_1$ under the assumption that β is $\mathsf{string}[10]$. Under this incorrect assumption, x has type $\mathsf{string}[10]$, but $ep1.2$ is actually a thirteen character string. Thus it seems that for tag comparisons the tag type should be at least invariant in its first argument (in fact, covariance is sound). Fortunately, the need for different subtyping behaviours arises in different situations: tag data structure creation and tag comparisons. Thus a variance mechanism (*c.f.* Abadi and Cardelli [AC96]) can be used to get each behaviour when it is needed.

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 combines the keys ideas of the previous section, tag types with a variance mechanism and physical pointer equality. The syntax is:

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

The operation $\mathsf{tag}(\langle\vec{e}\rangle, \tau)$ 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 $\mathsf{if}\ e_1 = e_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$. 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, option types, and a parallel let form. The

Syntax:

Values	v	$::=$	$x \mid \mathsf{roll}^\tau(v) \mid \mathsf{pack}[\tau_1, v]\ \mathsf{as}\ \tau_2 \mid \mathsf{none}^\tau \mid \mathsf{some}(v) \mid \mathsf{fix}\ f(x:\tau_1):\tau_2.b$
Contexts	E	$::=$	$[] \mid \mathsf{tag}(\langle \vec{v}, E, \vec{e}\rangle, \tau) \mid \mathsf{if}\ E = e\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid$ $\mathsf{if}\ v = E\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid \mathsf{roll}^\tau(E) \mid \mathsf{unroll}(E) \mid \mathsf{pack}[\tau_1, E]\ \mathsf{as}\ \tau_2 \mid$ $\mathsf{unpack}[\alpha, x] = E\ \mathsf{in}\ e \mid \mathsf{some}(E) \mid \mathsf{if?}\ E\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid$ $\langle \vec{v}, E, \vec{e}\rangle \mid E.i \mid E\ e \mid v\ E \mid \mathsf{let}\ x_1 = E\ \mathsf{and}\ x_2 = e_2\ \mathsf{in}\ e \mid$ $\mathsf{let}\ x_1 = v\ \mathsf{and}\ x_2 = E\ \mathsf{in}\ e$
Heap Values	h	$::=$	$\langle \vec{v}\rangle \mid \mathsf{tag}(\langle \vec{v}\rangle, \tau)$
Heaps	H	$::=$	$x_1 = h_1, \ldots, x_n = h_n$
Programs	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
$\langle \vec{v}\rangle$ or $\mathsf{tag}(\langle \vec{v}\rangle, \tau)$	x	$H, x = I$	x fresh
$\mathsf{if}\ x = x\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$	b_1	H	
$\mathsf{if}\ x = y\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$	b_2	H	$x \neq y$
$\mathsf{unroll}(\mathsf{roll}^\tau(v))$	v	H	
$\mathsf{unpack}[\alpha, x] = \mathsf{pack}[\tau_1, v]\ \mathsf{as}\ \tau_2\ \mathsf{in}\ e$	$e[\alpha := \tau_1, x := v]$	H	
$\mathsf{if?}\ \mathsf{none}^\tau\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$	b_2	H	
$\mathsf{if?}\ \mathsf{some}(v)\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$	$b_1[x := v]$	H	
$x.i$	v_i	H	$H(x) = v$ or $\mathsf{tag}(v, \tau)$
			$1 \leq i \leq n, v = \langle v_1, \ldots, v_n\rangle$
$v_1\ v_2$	$b[f, x := v_1, v_2]$	H	$v_1 = \mathsf{fix}\ f(x:\tau_1):\tau_2.b$
$\mathsf{let}\ x_1 = v_1\ \mathsf{and}\ x_2 = v_2\ \mathsf{in}\ e'$	$e'[x_1, x_2 := v_1, v_2]$	H	

Figure 4: Target Language Operational Semantics

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 $\mathsf{if?}\ e_1\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$ can be used to discriminate the two. The let form could be defined in terms of tuples and functions, but was included to ease proving operational correctness.

The α in $\mathsf{rec}\ \alpha.\tau$ and $\exists\alpha.\tau$ binds in τ; the α and x in $\mathsf{unpack}[\alpha, x] = e_1\ \mathsf{in}\ e_2$ binds in e_2; the x in $\mathsf{if?}\ e_1\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$ binds in b_1; the f and x in $\mathsf{fix}\ f(x{:}\tau_1){:}\tau_2.b$ bind in b; the x_1 and x_2 in $\mathsf{let}\ x_1 = e_1\ \mathsf{and}\ x_2 = e_2\ \mathsf{in}\ e$ bind in e.

The operational semantics is similar in spirit to the tagging language, and is given in Figure 4. Recall, that in an allocation style semantics, pointers are modeled by variables, and two pointers are equal if they are the same variable. Hence the rules for the if construct.

The static semantics appears in Figures 5 and 6. Note that a typing context of the form $\Delta, \alpha \leq \tau$ must satsify the condition in parenthesis in the figure. This syntactic restriction ensures that all typing contexts are well formed, simplying the proof of soundness. The static semantics is straightforward except for the tagging constructs. First, values in the type $\mathsf{tag}^\phi(\tau_1, \tau_2)$ also have type τ_2, and there is a specialised subsumption rule to reflect this. This rule is used to

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

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

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

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

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

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

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

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

Figure 5: Target Language Typing Rules for Types

manipulate the datastructure a type tag contains. There are two rules for tag comparison: one is for typing the translation and the other is for proving type preservation and soundness. Rule **t1** requires that e_1 be a tag for an unknown type, in fact, for a supertype of an unknown type. It also requires that e_2 be a tag for a known closed type, again, actually a subtype of the known type. I assume that, 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.

During the actual execution of translated code, the unknown type α is replaced by a known closed type. Therefore, to prove the preservation of typing across the reduction that replaces α, rule **t2** allows the comparison of two known types. If $\epsilon\vdash_{\mathcal{T}}\sigma_1\le\sigma_2$ 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 Appendix A. 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 $\epsilon\vdash_{\mathcal{T}}\sigma_1\le\sigma_2$, this is lemma A.9. Then by rule **t2** the then branch must type check. Another difficulty is with the type substitution lemma. In particular in proving that if $\alpha;\Gamma\vdash_{\mathcal{T}}\mathsf{if}\ e_1=e_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}:\tau$ and $\epsilon\vdash_{\mathcal{T}}\sigma$ then $\epsilon;\Gamma[\alpha:=\sigma]\vdash_{\mathcal{T}}\mathsf{if}\ e_1=e_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}[\alpha:=\sigma]:\tau[\alpha:=\sigma]$. If rule **t1** is used with $\alpha;\Gamma\vdash_{\mathcal{T}}e_1:\mathsf{tag}^{-}(\alpha,\tau_1)$ used to show the hypothesis then the conclusion has to follow from rule **t2**. There must be a certain coherence between these two rules.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6: Target Language Typing Rules for Terms

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 $\mathsf{rec}\ \alpha.\mathsf{tag}^-([\![\tau]\!]_{\mathrm{type}}, \langle \alpha? \rangle)$. 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, $\mathsf{tag}^\circ([\![\tau]\!]_{\mathrm{type}}, \langle (\mathsf{rec}\ \alpha.\mathsf{tag}^-([\![\tau]\!]_{\mathrm{type}}, \langle \alpha? \rangle))? \rangle)$, except the option type is shifted into the recursive type, $\mathsf{tag}^\circ([\![\tau]\!]_{\mathrm{type}}, \langle \mathsf{rec}\ \alpha.\mathsf{tag}^-([\![\tau]\!]_{\mathrm{type}}, \langle \alpha \rangle)? \rangle)$. The translation of tagged is just a pair of a tag for α and an α abstracted over α. The operations $\mathsf{newtag}(\tau)$, $\mathsf{subtag}(e, \tau)$, and $\mathsf{tagged}(e)$ 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 present 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. Proving type preservation is a straight forward inductive proof, and appears in Appendix B. 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 simulation argument. Most of the work is in showing that the tag check is implemented properly. The proof appears in Appendix C.

5 Extensions

The implementation given is quite naive, it takes time linear in the height of the tag hierarchy. Java implementations typically trade space for time, and represent tags as an array of all of the tags ancestors. This implementation requires only one array subscript and one physical pointer equality test, but uses space proportional to height of the tag hierarchy, and it is not suitable for multiple inheritance hierarchies. Appendix D gives an extension of the target language, and shows how to express this optimised implementation strategy in the extended language. I expect that most schemes for implementing multiple inheritance hierarchies efficiently could also be expressed in variants of the target language.

The particular target language chosen is not crucial to implementing type tagging mechanisms. In fact, the key ideas were used to augment TALx86 [MCG+99], a typed assembly language [MWCG98] for Intel's IA32 architecture, to provide support for downcasting in object oriented languages and support for exceptions. This implementation was straightforward:

$$
\begin{array}{lcl}
\mathit{tag}(\phi,\tau) & = & \mathsf{tag}^{\phi}(\tau,\langle \mathit{tag}'(\tau)\rangle) \\
\mathit{tag}'(\tau) & = & \mathsf{rec}\ \alpha.\mathsf{tag}^{-}(\tau,\langle\alpha\rangle)? \\
[\![\mathsf{tag}(\tau)]\!]_{\mathrm{type}} & = & \mathit{tag}(\circ,[\![\tau]\!]_{\mathrm{type}}) \\
[\![\mathsf{tagged}]\!]_{\mathrm{type}} & = & \exists\alpha.\langle \mathit{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{tag}(\langle\mathsf{roll}^{\mathit{tag}'([\![\tau]\!]_{\mathrm{type}})}(\mathsf{none}^{\mathit{tag}(-,[\![\tau]\!]_{\mathrm{type}})})\rangle,[\![\tau]\!]_{\mathrm{type}}) \\
[\![\mathsf{subtag}(e,\tau)]\!]_{\mathrm{exp}} & = & \mathsf{tag}(\langle\mathsf{roll}^{\mathit{tag}'([\![\tau]\!]_{\mathrm{type}})}(\mathsf{some}([\![e]\!]_{\mathrm{exp}}))\rangle,[\![\tau]\!]_{\mathrm{type}}) \\
[\![\mathsf{tagged}(e:\langle\mathsf{tag}(\tau),\tau\rangle)]\!]_{\mathrm{exp}} & = & \mathsf{pack}[[\![\tau]\!]_{\mathrm{type}},[\![e]\!]_{\mathrm{exp}}]\ \mathsf{as}\ [\![\mathsf{tagged}]\!]_{\mathrm{type}} \\
\mathit{tagchk}(y_1,y_2,\tau,\sigma) & = & \mathsf{fix}\ y_3(y_4:\mathit{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{iftagof}\ e_1=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?}\ \mathit{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{tag}(\langle\mathsf{roll}^{\mathit{tag}'([\![\tau]\!]_{\mathrm{type}})}(\mathsf{none}^{\mathit{tag}(-,[\![\tau]\!]_{\mathrm{type}})})\rangle,[\![\tau]\!]_{\mathrm{type}}) \\
[\![(\tau,x)]\!]_{\mathrm{hval}} & = & \mathsf{tag}(\langle\mathsf{roll}^{\mathit{tag}'([\![\tau]\!]_{\mathrm{type}})}(\mathsf{some}(x))\rangle,[\![\tau]\!]_{\mathrm{type}}) \\
[\![\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

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.

6 Related Work

This work is closely related to work on type dispatch in functional languages [HM95, Mor95, CWM98, CW99, *etc.*]. In particular, the work on λ^R [CWM98] and its follow-up LX [CW99] both address the implementation of λ_i^{ML} type dispatch. There are two key differences between their work and mine. 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. Furthermore, their language does not handle subtyping. Second, my language is concerned with branded types whereas λ^R is concerned with the types themselves. Consider my informal coding of exceptions. In Ocaml 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. Of course, λ^R is designed to implement intensional polymorphism and does that well, whereas my language cannot implement intensional polymorphism.

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 functional languages, 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.

[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. Technical report, Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA, 1999. Forthcoming, available at `http://www.cs.cornell.edu/sweirich`.

[CWM98] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *Third 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999. To appear. 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.

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

[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 type assembly language. February 1999. Submitted to WCSSS'99, available at `http://www.cs.cornell.edu/talc`.

[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 *Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego California, USA, January 1998.

[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.

A Type Soundness of Target Calculus

The presence of the two subsumption rules and reflexive and transitive subtyping rules complicates the arguments for type soundness. Thus to start with, I will argue that derivations in the systems can be canonicalised and then argue based on the canonical derivations. First consider subtyping, and a more syntax directed set of subtyping rules.

Let S_1 be the subtyping rules in Figure 5. Let S_2 be S_1 with the reflexive rule specialised to type variables and recursive types only, the transitive rule removed, and the rule for type variables replaced by:

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

Lemma A.1 *The judgement $\Delta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2$ is derivable in S_1 if and only if it is derivable in S_2.*

Proof: First I will show that S_2 is reflexive and transitive, that is, that the general reflexive and the transitive rules are derivable. The proof for reflexivity proceeds by induction on the structure of τ. In the cases that τ is a type variable or a recursive type, the desired result follows by the specialised reflexive rules. In other cases the induction hypothesis is applied to the immediate subterms and then the one rule for that case is applied. The proof for transitivity proceeds by simultaneous induction on derivations of $\Delta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2$ and $\Delta \vdash_{\mathcal{T}} \tau_2 \leq \tau_3$. If either judgement was derived by the specialised reflexive rules, then the result is immediate. Otherwise, enumerate the cases for the last rule used in each derivation and notice that either the same rule is used or the variable rule is used. In the case that the same rule is used, apply the induction hypothesis to the hypotheses in the derivations and then apply the same rule again. In the case the type variable rule is used on $\Delta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2$, apply the induction hypothesis to the hypothesis of this derivation and the other judgement and then apply the type rule again. In the case the type variable rule is used on $\Delta \vdash_{\mathcal{T}} \tau_2 \leq \tau_3$, it must be that τ_2 is a type variable, so the only rule for deriving $\Delta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2$, which has a type variable on the right hand side, is the type variable rule so the previous case applies.

Now I will show that a judgement derivable in S_1 is derivable in S_2. I have just shown that the reflexive and transitive rules are derivable. The other rules except the one for type variables are rules of S_2. The type variable rule of S_1 can be derived in S_2 by applying the derived reflexive rule and then the type variable rule.

Finally I will show that a judgement derivable in S_2 is derivable in S_1. All the rules of S_2 are rules of S_1 except for the type variable rule, this can be derived by applying S_1's type variable rule and then the transitive rule. $\square$

From now on, I will use both systems interchangeably.

Second, in a program derivation, I can assume that the derivation of a heap value ends in either the tag rule or the tuple rule. The only other applicable rules are the tag subsumption rule and the subsumption rule. These rules can be removed and then inserted at all places in the derivation to the right where the variable in question appears.

Third, in any expression derivation between any two uses of rules that are not tag subsumption or subsumption rules I can assume there is exactly one use of the subsumption rule and zero or more uses of the tag subsumption rule. Furthermore, it will often be the case the there are zero uses of the tag subsumption rule because the types in question will not have a tag form. The assumption is valid because the tag subsumption and subsumption clearly commute, two consequtive uses of the subsumption rule can be replaced with one by using the transitive rule, and a subsumption rule can be inserted by using the reflexive rule.

With these arguments about subtyping and about where subsumption and tag subsumption rules can appear, I proceed to show type soundness of the target calculus, through a series of lemmas.

Typing context Δ_1 *extends* Δ_2 if and only if the type variables defined by Δ_2 all appear in Δ_1 and in the same order, and either Δ_2 does not give the type variable a bound or Δ_1 gives it the same bound. Value context Γ_1 extends Γ_2 if and only if $\mathrm{dom}(\Gamma_1) \subseteq \mathrm{dom}(\Gamma_2)$ and for all $x \in \mathrm{dom}(\Gamma_2)$ $\Gamma_1(x) = \Gamma_2(x)$. Note that the value context used to type the main expression of a program extends all the value contexts used to type the heap values in the program.

Lemma A.2 (Weakening) *If Δ_1 extends Δ_2 and Γ_1 extends Γ_2 then:*

1. *If $\Delta_2 \vdash_{\mathcal{T}} \tau$ then $\Delta_1 \vdash_{\mathcal{T}} \tau$*
2. *If $\Delta_2 \vdash_{\mathcal{T}} \tau_1 \leq \tau_2$ then $\Delta_1 \vdash_{\mathcal{T}} \tau_1 \leq \tau_2$*
3. *If $\Delta_2; \Gamma_2 \vdash_{\mathcal{T}} e : \tau$ then $\Delta_1; \Gamma_1 \vdash_{\mathcal{T}} e : \tau$*

Proof: The first result is immediate from the definitions. The proof of the other results proceeds by induction on the structure of the hypothesis derivation and the desired result follows in a straightforward way from the induction hypothesis on the hypotheses of the last rule used in the derivation. $\square$

Lemma A.3 (Derived Judgements)

1. *If $\Delta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2$ then $\Delta \vdash_{\mathcal{T}} \tau_1$ and $\Delta \vdash_{\mathcal{T}} \tau_2$.*

Proof: The proof proceeds by induction on the derivation of $\Delta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2$ and the result follows in a straightforward way from the induciton hypothesis on the hypotheses of the last rule used in the derivation, or in the case of the reflexive and recursive type rules, directly from a hypothesis. Note that the syntactic restriction on typing contexts is used in the case of the typing variable rule. □

Lemma A.4 (Type Substitution) *If $\alpha \notin \Delta_1$ then:*

1. *If $\Delta_1 \vdash_{\mathcal{T}} \sigma$ and $\Delta_1, \alpha, \Delta_2 \vdash_{\mathcal{T}} \tau$ or $\Delta_1, \alpha \leq \sigma', \Delta_2 \vdash_{\mathcal{T}} \tau$ then $\Delta_1, \Delta_2[\alpha := \sigma] \vdash_{\mathcal{T}} \tau[\alpha := \sigma]$.*
2. *If $\Delta_1 \vdash_{\mathcal{T}} \sigma$ and $\Delta_1, \alpha, \Delta_2 \vdash_{\mathcal{T}} \tau_1 \leq \tau_2$ then $\Delta_1, \Delta_2[\alpha := \sigma] \vdash_{\mathcal{T}} \tau_1[\alpha := \sigma] \leq \tau_2[\alpha := \sigma]$.*
3. *If $\Delta_1 \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_2$ and $\Delta_1, \alpha \leq \sigma_2, \Delta_2 \vdash_{\mathcal{T}} \tau_1 \leq \tau_2$ then $\Delta_1, \Delta_2[\alpha := \sigma_1] \vdash_{\mathcal{T}} \tau_1[\alpha := \sigma_1] \leq \tau_2[\alpha := \sigma_1]$.*
4. *If $\epsilon \vdash_{\mathcal{T}} \sigma$ and $\Delta_1, \alpha, \Delta_2; \Gamma \vdash_{\mathcal{T}} e : \tau$ then $\Delta_1, \Delta_2[\alpha := \sigma]; \Gamma[\alpha := \sigma] \vdash_{\mathcal{T}} e[\alpha := \sigma] : \tau[\alpha := \sigma]$.*
5. *If $\epsilon \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_2$ and $\Delta_1, \alpha \leq \sigma_2, \Delta_2; \Gamma \vdash_{\mathcal{T}} e : \tau$ then $\Delta_1, \Delta_2[\alpha := \sigma_1]; \Gamma[\alpha := \sigma_1] \vdash_{\mathcal{T}} e[\alpha := \sigma_1] : \tau[\alpha := \sigma_1]$.*

Proof: The proof of the first result is immediate from the definitions.

The proof of the second result proceeds by induction on the derivation in S_2 of $\Delta, \alpha \vdash_{\mathcal{T}} \tau_1 \leq \tau_2$. Apply the induction hypothesis to the hypotheses of the last rule used in the derivation and then reapply the rule. This works in every case except if the last rule is the reflexive rule for α, in this case $\Delta \vdash_{\mathcal{T}} \alpha[\alpha := \sigma] \leq \alpha[\alpha := \sigma]$ follows by the reflexive rule, $\Delta_1 \vdash_{\mathcal{T}} \sigma$, and the weakening lemma.

The proof of the third result is similar except if the last rule used is new type variable rule for α. In this case $\Delta_1, \alpha \leq \sigma_2, \Delta_2 \vdash_{\mathcal{T}} \alpha \leq \tau_2$, and the hypothesis of the last rule is $\Delta_1, \alpha \leq \sigma_2, \Delta_2 \vdash_{\mathcal{T}} \sigma_2 \leq \tau_2$. Applying the induction hypothesis, it must be that $\Delta_1, \Delta_2[\alpha := \sigma_1] \vdash_{\mathcal{T}} \sigma_2[\alpha := \sigma_1] \leq \tau_2[\alpha := \sigma_1]$. By the derived judgements lemma, $\Delta_1 \vdash_{\mathcal{T}} \sigma_2$, so $\alpha \notin \mathrm{ftv}(\sigma_2)$, and thus $\Delta_1, \Delta_2[\alpha := \sigma_1] \vdash_{\mathcal{T}} \sigma_2 \leq \tau_2[\alpha := \sigma_1]$. By the we the weakening lemma $\Delta_1, \Delta_2[\alpha := \sigma_1] \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_2$, so by tranisitivity $\Delta_1, \Delta_2[\alpha := \sigma_1] \vdash_{\mathcal{T}} \alpha[\alpha := \sigma_1] \leq \tau_2[\alpha := \sigma_1]$ as required.

The proof of the fourth and fifth results proceed by induction of the derivation of $C \vdash_{\mathcal{T}} e : \tau$ (for appropriate C). In almost all cases the desired result follows in a straight forward manner from the induction hypothesis applied to the hypotheses of the last rule used in the derivation and from (1), (2) and (3) applied to typing hypotheses. The only interesting case is rule **t1** when α in that rule is the same as the α in question. Applying the induction hypothesis to the first three and last hypothesis gives the first, third, fourth, and sixth hypotheses of rule **t2**. If $\epsilon \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_2$ then by the induction hypothesis applied to the fourth hypothesis, $\Delta_1, \Delta_2[\alpha := \sigma_1]; \Gamma[\alpha := \sigma_2] \vdash_{\mathcal{T}} b_2[\alpha := \sigma_2] : \tau[\alpha := \sigma_1]$, which is the fifth hypothesis of rule **t2**. The second hypothsis follows from the derived judgements lemma, and rule **t2** gives the desired result. □

Note that in typing derivations for programs, the α in Δ are always distinct, because Δ starts out empty and the rules only add α not already in Δ. Thus the condition $\alpha \notin \Delta_1$ will always be satsified for judgements that come from program derivations.

Lemma A.5 *If* $\Delta \vdash_{\mathcal{T}} \mathsf{rec}\ \alpha.\tau_1 \leq \mathsf{rec}\ \alpha.\tau_2$ *then* $\Delta \vdash_{\mathcal{T}} \tau_1[\alpha := \mathsf{rec}\ \alpha.\tau_1] \leq \tau_2[\alpha := \mathsf{rec}\ \alpha.\tau_2]$.

Proof: In S_2 there are two rules for $\Delta \vdash_{\mathcal{T}} \mathsf{rec}\ \alpha.\tau_1 \leq \mathsf{rec}\ \alpha.\tau_2$: the reflexive rule for recursive types or the rule for recursive types. In the case of the reflexive rule, $\tau_1 = \tau_2$ so $\tau_1[\alpha := \mathsf{rec}\ \alpha.\tau_1] = \tau_2[\alpha := \mathsf{rec}\ \alpha.\tau_2]$ and the desired result follows by the reflexive rule in S_1. In the case of the recursive types rule, it must be that $\Delta, \alpha \vdash_{\mathcal{T}} \tau_1 \leq \tau_2$. The desired result follows by the type substitution lemma. □

Lemma A.6 (Decomposition) *An expression e is either a value or of the form $E[\iota]$ for some E and some ι drawn from the following grammar:*

$$\begin{array}{rcl} \iota & ::= & h \mid \mathsf{if}\ v_1 = v_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid \mathsf{unroll}(v) \mid \mathsf{unpack}[\alpha, x] = v\ \mathsf{in}\ e \mid \\ & & \mathsf{if?}\ v\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid v.i \mid v_1\ v_2 \mid \mathsf{let}\ x_1 = v_2\ \mathsf{and}\ x_2 = v_2\ \mathsf{in}\ e \end{array}$$

Proof: The proof proceeds by induction on the structure of e and a massive inspection of the various forms for e and E. □

Lemma A.7 (Context Typing) *If* $\epsilon; \Gamma \vdash_{\mathcal{T}} E[\iota] : \tau$ *then there exists* σ *such that* $\epsilon; \Gamma \vdash_{\mathcal{T}} \iota : \sigma$ *and for all* e *if* $\epsilon; \Gamma, \Gamma' \vdash_{\mathcal{T}} e : \sigma$ *then* $\epsilon; \Gamma, \Gamma' \vdash_{\mathcal{T}} E[e] : \tau$. *Furthermore, the derivation of* $\epsilon; \Gamma \vdash_{\mathcal{T}} \iota : \sigma$ *does not end in a use of either the subsumption or tag subsumption rule.*

Proof: The proof proceeds by induction on the derivation of $\epsilon; \Gamma \vdash_{\mathcal{T}} E[\iota] : \tau$. The desired result follows straightforwardly from the induction hypothesis on the appropriate hypothesis of the last rule in the derivation. □

A value typing Γ has a heap form if and only if $\forall \tau \in \mathrm{ran}(\Gamma) : \tau = \langle \vec{\tau} \rangle$ or $\mathsf{tag}^{\phi}(\langle \tau \rangle, \sigma)$. Note that if $\vdash_{\mathcal{T}} H : \Gamma$ then Γ has a heap form, hence the name.

Lemma A.8 (Canonical Forms) *If* Γ *has a heap form and* $\epsilon; \Gamma \vdash_{\mathcal{T}} v : \tau$ *then* v *has the form given by the following table:*

τ	v
$\mathsf{tag}^{\phi}(\tau_1, \tau_2)$	x
$\mathsf{rec}\ \alpha.\sigma$	$\mathsf{roll}^{\mathsf{rec}\ \alpha.\sigma'}(v')$
$\exists \alpha.\sigma$	$\mathsf{pack}[\tau', v']\ \mathsf{as}\ \exists \alpha.\sigma'$
$\sigma?$	$\mathsf{none}^{\sigma'}$ *or* $\mathsf{some}(v')$
$\langle \vec{\sigma} \rangle$	x
$\sigma_1 \rightarrow \sigma_2$	$\mathsf{fix}\ f(x : \tau_1) : \tau_2.v$

Proof: The derivation of $\epsilon; \Gamma \vdash_{\mathcal{T}} v : \tau$ consists of a nonsubsumption rule followed by an optional tag subsumption rule and an optional subsumption rule. By inspection of S_2 with an empty typing context note that all the applicable rules for subsumption do not change the form of τ. Next, notice that tag subsumption changes τ from a tuple form to a tag form. Now, inspect the rules for typing values for each of the forms for τ and notice that only one or two rules are applicable in each case and have the forms shown for v in the table. □

Lemma A.9 *If* $\epsilon; \Gamma \vdash_{\mathcal{T}} x : \mathsf{tag}^{-}(\sigma_1, \tau_1)$ *and* $\epsilon; \Gamma \vdash_{\mathcal{T}} x : \mathsf{tag}^{+}(\sigma_2, \tau_2)$ *then* $\epsilon \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_2$.

Proof: The only rules for deriving the judgements are the variable rule and the subsumption rule, therefore there must be a τ such that $\epsilon \vdash_{\mathcal{T}} \tau \leq \mathsf{tag}^{-}(\sigma_1, \tau_1)$ and $\epsilon \vdash_{\mathcal{T}} \tau \leq \mathsf{tag}^{+}(\sigma_2, \tau_2)$. In S_2 the only rules that can derive these judgements are the tag subtyping rules, so it must be the case that $\tau = \mathsf{tag}^{\circ}(\sigma, \tau)$ and that $\epsilon \vdash_{\mathcal{T}} \sigma_1 \leq \sigma$ and $\epsilon \vdash_{\mathcal{T}} \sigma \leq \sigma_2$. The desired result follows by the transitivity rule. □

Lemma A.10 (Value Substitution) *If* $\Delta; \Gamma_1 \vdash_{\mathcal{T}} e_1 : \tau_1$ *and* $\Delta; \Gamma_1, x : \tau_1, \Gamma_2 \vdash_{\mathcal{T}} e_2 : \tau_2$ *then* $\Delta; \Gamma_1, \Gamma_2 \vdash_{\mathcal{T}} e_2[x := e_1] : \tau_2$.

Proof: The proof proceeds by induction on the derivation of $\Delta; \Gamma, x : \tau_1 \vdash_{\mathcal{T}} e_2 : \tau_2$. The desired result follows in a striaghforward manner from the induction hypothesis applied to the hypotheses of the last rule in the derivation. □

Theorem A.11 (Type Preservation) *If* $\vdash_{\mathcal{T}} P_1 : \tau$ *and* $P_1 \mapsto P_2$ *then* $\vdash_{\mathcal{T}} P_2 : \tau$.

Proof: Let $P_1 = \mathsf{let}\ H_1\ \mathsf{in}\ E[\iota]$ and $P_2 = \mathsf{let}\ H_2\ \mathsf{in}\ E[e]$ where ι, e, and H_2 are given by one of the rules in Figure 4. The derivation of $\vdash_{\mathcal{T}} P_1 : \tau$ must be by the rule for programs, so there exists Γ_1 such that $\vdash_{\mathcal{T}} H_1 : \Gamma_1$ and $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} E[\iota] : \tau$. So by lemma A.7 there exists σ such that $\Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma$. The goal is to find Γ_2 such that $\vdash_{\mathcal{T}} H_2 : \Gamma_1, \Gamma_2$ and $\epsilon; \Gamma_1, \Gamma_2 \vdash_{\mathcal{T}} e : \sigma$. Then by lemma A.7 $\epsilon; \Gamma_1, \Gamma_2 \vdash_{\mathcal{T}} E[e] : \tau$ and by the program rule $\vdash_{\mathcal{T}} P_2 : \tau$ as desired. The proof proceeds by a case analysis of the rule used for $P_1 \mapsto P_2$.

$\iota = \langle \vec{v} \rangle$ **or** $\mathsf{tag}(\vec{v}, \tau)$**:** In this case $e = x$ for some fresh x and $H_2 = H_1, x = \iota$. Clearly, by the rule for heaps, $\vdash_{\mathcal{T}} H_1, x = \iota : \Gamma_1, x : \sigma$. So selecting $\Gamma_2 = x : \sigma$, it remains to show that $\Gamma_1, \Gamma_2 \vdash_{\mathcal{T}} x : \sigma$ which follows by the variable rule.

$\iota = \mathsf{if}\ x = x\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$**:** In this case $e = b_1$ and $H_2 = H_1$. Selecting $\Gamma_2 = \epsilon$ it remains to show that $\Gamma_1 \vdash_{\mathcal{T}} b_1 : \tau$. Since ι type checks in an empty type context, only rule **t2** is applicable. Therefore the derivation of $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma$ has the form:

$$\frac{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} x : \mathsf{tag}^{-}(\sigma_1, \tau_1) \quad \epsilon; \Gamma_1 \vdash_{\mathcal{T}} x : \mathsf{tag}^{+}(\sigma_2, \tau_2) \quad \begin{array}{c}\epsilon \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_2 \\ \Rightarrow \\ \epsilon; \Gamma_1 \vdash_{\mathcal{T}} b_1 : \sigma\end{array} \quad \cdots}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \mathsf{if}\ x = x\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} : \sigma}$$

By the two judgements on x and lemma A.9 it must be that $\epsilon \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_2$. By the judgement on b_1, $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} b_1 : \sigma$ as required.

$\iota = \mathsf{if}\ x = y\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$**:** In this case $e = b_2$ and $H_2 = H_1$. Selecting $\Gamma_2 = \epsilon$ it remains to show that $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} b_2 : \sigma$. By similar reasoning to the previous case, $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma$ has the form:

$$\frac{\cdots \quad \epsilon; \Gamma_1 \vdash_{\mathcal{T}} b_2 : \sigma}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \mathsf{if}\ x = x\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} : \sigma}$$

The desired result is the judgement on b_2.

$\iota = \mathsf{unroll}(\mathsf{roll}^{\sigma_1}(v))$**:** In this case $e = v$ and $H_2 = H_1$. Selecting $\Gamma_2 = \epsilon$, it remains to show that $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} e : \sigma$. The derivation of $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma$ has the form:

$$\frac{\dfrac{\dfrac{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} v : \sigma_2[\alpha := \sigma_1]}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \mathsf{roll}^{\sigma_1}(v) : \sigma_1} \quad \epsilon \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_3}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \mathsf{roll}^{\sigma_1}(v) : \sigma_3}}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma}$$

where $\sigma = \sigma_4[\alpha := \sigma_3]$, $\sigma_1 = \mathsf{rec}\ \alpha.\sigma_2$, and $\sigma_3 = \mathsf{rec}\ \alpha.\sigma_4$. Note that the tag sumsumption rule cannot be used between the roll and unroll rule because the types have a recursive form not a tag form. By $\epsilon \vdash_{\mathcal{T}} \sigma_1 \leq \sigma_3$ and lemma A.5, $\epsilon \vdash_{\mathcal{T}} \sigma_2[\alpha := \sigma_1] \leq \sigma_3[\alpha := \sigma_4]$. The later is σ, so by judgement on v and subsumption $\epsilon; \Gamma \vdash_{\mathcal{T}} v : \sigma$ as required.

$\iota = \mathsf{unpack}[\alpha, c] = \mathsf{pack}[\tau_1, v]\ \mathsf{as}\ \tau_2\ \mathsf{in}\ b$**:** In this case $e = b[\alpha := \tau_1, x := v]$ and $H_2 = H_1$. Selecting $\Gamma_2 = \epsilon$, it remains to show that $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} e : \sigma$. The derivation of $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma$ has the form:

$$\frac{\dfrac{\dfrac{\epsilon \vdash_{\mathcal{T}} \tau_1 \quad \epsilon; \Gamma_1 \vdash_{\mathcal{T}} v : \tau_3[\alpha := \tau_1]}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \mathsf{pack}[\tau_1, v]\ \mathsf{as}\ \tau_2 : \tau_2} \quad \epsilon \vdash_{\mathcal{T}} \tau_2 \leq \exists \alpha.\tau_4}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \mathsf{pack}[\tau_1, v]\ \mathsf{as}\ \tau_2 : \exists \alpha.\tau_4} \quad \alpha; \Gamma_1, x : \tau_4 \vdash_{\mathcal{T}} b : \sigma \quad \epsilon \vdash_{\mathcal{T}} \sigma}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma}$$

where $\tau_2 = \exists \alpha.\tau_3$. Note that the tag subsumption rule cannot be used between the pack and unpack rules because the types have an existential form not a tag form. In S_2 there is only one rule for $\epsilon \vdash_{\mathcal{T}} \exists \alpha.\tau_3 \leq \exists \alpha.\tau_4$ so it must be that $\alpha \vdash_{\mathcal{T}} \tau_3 \leq \tau_4$. By the type substitution lemma, $\epsilon \vdash_{\mathcal{T}} \tau_3[\alpha := \tau_1] \leq \tau_4[\alpha = \tau_1]$. By subsumption $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} v : \tau_4[\alpha := \tau_1]$. Now, by the type substitution lemma, $\epsilon; (\Gamma_1, x : \tau_4)[\alpha := \tau_1] \vdash_{\mathcal{T}} b[\alpha := \tau_1] : \sigma[\alpha := \tau_1]$. Since Γ_1 is the type for H_1 it has no free type variables so $(\Gamma_1, x : \tau_4)[\alpha := \tau_1] = \Gamma_1, x : \tau_4[\alpha := \tau_1]$, and $\epsilon \vdash_{\mathcal{T}} \sigma$ implies $\mathrm{ftv}(\sigma) = \emptyset$ so $\sigma[\alpha := \tau_1] = \sigma$. So $\epsilon; \Gamma_1, x : \tau_4[\alpha := \tau_1] \vdash_{\mathcal{T}} b[\alpha := \tau_1] : \sigma$. By the value substitution lemma $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} b[\alpha := \tau_1, x := v] : \sigma$ as required.

$\iota = \mathsf{if?}\ \mathsf{none}^{\tau'}\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$**:** In this case $e = b_2$ and $H_2 = H_1$. Selecting $\Gamma_2 = \epsilon$ it remains to show that $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} e : \sigma$. The derivation of $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma$ must have the form:

$$\frac{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \mathsf{none}^{\tau'} : \tau'? \quad \epsilon; \Gamma_1, x : \tau' \vdash_{\mathcal{T}} b_1 : \sigma \quad \epsilon; \Gamma_1 \vdash_{\mathcal{T}} b_2 : \sigma}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma}$$

Note that the tag subsumption rule cannot be used between the none rule and the opt if rule because the types have an opt form not a tag form. The judgemnet on b_2 is the desired result.

$\iota = \mathsf{if?}\ \mathsf{some}(v)\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$**:** In this case $e = b_1[x := v]$ and $H_2 = H_1$. Selecting $\Gamma_2 = \epsilon$ it remains to show that $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} e : \sigma$. The derivation of $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma$ must have the form:

$$\frac{\dfrac{\dfrac{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} v : \tau_2}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \mathsf{some}(v) : \tau_2?} \quad \epsilon \vdash_{\mathcal{T}} \tau_2? \leq \tau_1?}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \mathsf{some}(v) : \tau_1?} \quad \epsilon; \Gamma_1, x : \tau_1 \vdash_{\mathcal{T}} b_1 : \sigma \quad \epsilon; \Gamma_1 \vdash_{\mathcal{T}} b_2 : \sigma}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma}$$

Note that the tag subsumption rule cannot be used between the some rule and the opt if rule because the types have an opt form not a tag form. In S_2 there is only one rule for deriving $\epsilon \vdash_{\mathcal{T}} \tau_2? \leq \tau_1?$, so it must be that $\epsilon \vdash_{\mathcal{T}} \tau_2 \leq \tau_1$. By the judgement on v and subsumption $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} v : \tau_1$. By the judgement on b_1 and the value substitution lemma $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} b_1[x := v]\sigma :$ as required.

$\iota = x.i$**:** In this case $e = v_i$ and $H_2 = H_1$. Selecting $\Gamma_2 = \epsilon$ it remains to show that $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} e : \sigma$. There are two cases: $H(x) = \langle v_1, \ldots, v_n \rangle$ or $H(x) = \mathsf{tag}(\langle v_1, \ldots, v_n \rangle, \tau')$. Consider the former first. The derivation of $\vdash_{\mathcal{T}} H_1 : \Gamma_1$ must have the form:

$$\dfrac{\cdots \quad \dfrac{\epsilon; \Gamma_i \vdash_{\mathcal{T}} v_i : \tau_i}{\epsilon; \Gamma' \vdash_{\mathcal{T}} \langle v_1, \ldots, v_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle} \quad \cdots}{\vdash_{\mathcal{T}} \ldots, x = \langle v_1, \ldots, v_n \rangle, \ldots : \Gamma_1}$$

for some Γ' that Γ_1 extends. The derivation of $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma$ must have the form:

$$\dfrac{\dfrac{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} x : \langle \tau_1, \ldots, \tau_n \rangle \quad \epsilon \vdash_{\mathcal{T}} \langle \tau_1, \ldots, \tau_m \rangle \leq \langle \sigma_1, \ldots, \sigma_n \rangle}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} x : \langle \sigma_1, \ldots, \sigma_m \rangle}}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma}$$

where $\sigma = \sigma_i$. Note that the tag subsumption rule cannot be used between the tuple rule and the projection rule because the types have a tuple form not a tag form. In S_2 there is only one rule for $\epsilon \vdash_{\mathcal{T}} \langle \tau_1, \ldots, \tau_m \rangle \leq \langle \sigma_1, \ldots, \sigma_n \rangle$, so it must be that $\epsilon \vdash_{\mathcal{T}} \tau_i \leq \sigma_i$. Thus by the judgement for v_i and subsumption $\epsilon; \Gamma_i \vdash_{\mathcal{T}} v_i : \sigma$. By the weakening lemma $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} v_i : \sigma$ as required.

Now consider the case $H(x) = \mathsf{tag}(\langle v_1, \ldots, v_n \rangle, \tau')$. Now $\vdash_{\mathcal{T}} H_1 : \Gamma_1$ will have the form:

$$\dfrac{\cdots \quad \dfrac{\epsilon; \Gamma_i \vdash_{\mathcal{T}} v_i : \tau_i \quad \epsilon \vdash_{\mathcal{T}} \tau'}{\epsilon; \Gamma_i \vdash_{\mathcal{T}} \mathsf{tag}(\langle v_1, \ldots, v_n \rangle, \tau') : \mathsf{tag}^{\circ}(\tau', \langle \tau_1, \ldots, \tau_n \rangle)} \quad \cdots}{\vdash_{\mathcal{T}} \ldots, x = \mathsf{tag}(\langle v_1, \ldots, v_n \rangle, \tau'), \ldots : \Gamma_1}$$

and the derivation of $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} x : \langle \tau_1, \ldots, \tau_n \rangle$ has an extra rule:

$$\dfrac{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} x : \mathsf{tag}^{\circ}(\tau', \langle \tau_1, \ldots, \tau_n \rangle)}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} x : \langle \tau_1, \ldots, \tau_n \rangle}$$

Otherwise the reasoning is the same.

$\iota = v_1\ v_2$: Note that $v_1 = \mathsf{fix}\, f(x : \tau_1) : \tau_2.b$. In this case $e = b[f, x := v_1, v_2]$ and $H_2 = H_1$. Selecting $\Gamma_2 = \epsilon$ it remains to show that $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} e : \sigma$. The derivation of $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma$ must have the form:

$$\dfrac{\dfrac{\dfrac{\epsilon; \Gamma_1, f{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash_{\mathcal{T}} b : \tau_2}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} v_1 : \tau_1 \to \tau_2} \qquad \epsilon \vdash_{\mathcal{T}} \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} v_1 : \sigma_1 \to \sigma} \qquad \epsilon; \Gamma_1 \vdash_{\mathcal{T}} v_2 : \sigma_1}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma}$$

Note that the tag subsumption rule cannot be used between the fix rule and the application rule because the types have a function form not a tag form. In S_2 there is only one rule for $\epsilon \vdash_{\mathcal{T}} \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma$, so it must be that $\epsilon \vdash_{\mathcal{T}} \sigma_1 \leq \tau_1$ and $\epsilon \vdash_{\mathcal{T}} \tau_2 \leq \sigma$. Using the former, the judgement for v_2 and subsumption, $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} v_2 : \tau_1$. Then, by the value subsitution lemma and the judgements for v_1 and b, $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} b[f, x := v_1, v_2] : \tau_2$. Finally, using $\epsilon \vdash_{\mathcal{T}} \tau_2 \leq \sigma$ and subsumption the desired result follows.

$\iota = \mathsf{let}\, x_1 = v_1\, \mathsf{and}\, x_2 = v_2\, \mathsf{in}\, e'$: In this case $e = e'[x_1, x_2 := v_1, v_2]$ and $H_2 = H_1$. Selecting $\Gamma_2 = \epsilon$ it remains to show that $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} e : \sigma$. The derivation of $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma$ must have the form:

$$\dfrac{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} v_1 : \tau_1 \quad \epsilon; \Gamma_1 \vdash_{\mathcal{T}} v_2 : \tau_2 \quad \epsilon; \Gamma_1, x_1{:}\tau_1, x_2{:}\tau_2 \vdash_{\mathcal{T}} e' : \sigma}{\epsilon; \Gamma_1 \vdash_{\mathcal{T}} \iota : \sigma}$$

By the value substitution lemma and the judgements for v_2 and e', $\epsilon; \Gamma_1, x_1{:}\tau_1 \vdash_{\mathcal{T}} e'[x_2 := v_2] : \sigma$. By the value substitution lemma and the judgement for v_1, $\epsilon; \Gamma_1 \vdash_{\mathcal{T}} e'[x_1, x_2 := v_1, v_2] : \sigma$, as required.

□

A *terminal configuration* is a program of the form $\mathsf{let}\ H\ \mathsf{in}\ v$. A program that is irreducible and not a terminal configuration is called *stuck*.

Theorem A.12 (Progress) *If $\vdash_{\mathcal{T}} P : \tau$ then either P is a terminal configuration or there exists a P' such that $P \mapsto P'$.*

Proof: Let $P = \mathsf{let}\ H\ \mathsf{in}\ e$, then by the decomposition lemma either e is a value in which case P is a terminal configuration as required or $e = E[\iota]$ for some ι as given in that lemma. The derivation of $\vdash_{\mathcal{T}} P : \tau$ must be by the rule for programs, so there exists Γ such that $\vdash_{\mathcal{T}} H : \Gamma$ and $\epsilon; \Gamma \vdash_{\mathcal{T}} E[\iota] : \tau$. Note that the former implies that Γ has a heap form. By the latter and lemma A.7 there exists σ such that $\Gamma \vdash_{\mathcal{T}} \iota : \sigma$. I will show that the latter implies that ι has one of the forms for I in Figure 4 and that the side conditions are satisfied. The proof proceeds by a case analysis of the form of ι:

$\iota = h$: In this case ι has the first form in the table and clearly a fresh x can be chosen.

$\iota = \mathsf{if}\ v_1 = v_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$: The derivation of $\epsilon; \Gamma \vdash_{\mathcal{T}} \iota : \sigma$ must have the form:

$$\frac{\cdots \quad \epsilon; \Gamma \vdash_{\mathcal{T}} v_1 : \mathsf{tag}^-(\sigma_1, \tau_1) \quad \epsilon; \Gamma \vdash_{\mathcal{T}} v_2 : \mathsf{tag}^+(\sigma_2, \tau_2) \quad \cdots}{\epsilon; \Gamma \vdash_{\mathcal{T}} \iota : \sigma}$$

By the canonical forms lemma case (1) v_1 and v_2 are variables and therefore ι has either the second or the third form for I as required.

$\iota = \mathsf{unroll}(v)$: The derivation of $\epsilon; \Gamma \vdash_{\mathcal{T}} \iota : \sigma$ must have the form:

$$\frac{\epsilon; \Gamma \vdash_{\mathcal{T}} v : \mathsf{rec}\ \alpha.\sigma'}{\epsilon; \Gamma \vdash_{\mathcal{T}} \iota : \sigma}$$

By the canonical forms lemma case (2) v has the form $\mathsf{roll}^{\mathsf{rec}\ \alpha.\sigma''}(v')$ and therefore ι has the fourth form for I as required.

$\iota = \mathsf{unpack}[\alpha, x] = v\ \mathsf{in}\ b$: The derivation of $\epsilon; \Gamma \vdash_{\mathcal{T}} \iota : \sigma$ must have the form:

$$\frac{\epsilon; \Gamma \vdash_{\mathcal{T}} v : \exists\alpha.\sigma' \quad \cdots}{\epsilon; \Gamma \vdash_{\mathcal{T}} \iota : \sigma}$$

By the canonical forms lemma case (3) v has the form $\mathsf{pack}[\tau', v']\ \mathsf{as}\ \exists\alpha.\sigma''$, therefore ι has the fifth form for I as required.

$\iota = \mathsf{if?}\ v\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$: The derivation of $\epsilon; \Gamma \vdash_{\mathcal{T}} \iota : \sigma$ must have the form:

$$\frac{\epsilon; \Gamma \vdash_{\mathcal{T}} v : \sigma'? \quad \cdots}{\epsilon; \Gamma \vdash_{\mathcal{T}} \iota : \sigma}$$

By the canonical forms lemma case (4) v has the form $\mathsf{none}^{\sigma''}$ or $\mathsf{some}(v')$, therefore ι has either the sixth or seventh form for I as required.

$\iota = v.i$**:** The derivation of $\epsilon; \Gamma \vdash_{\mathcal{T}} \iota : \sigma$ must have the form:

$$\frac{\epsilon; \Gamma \vdash_{\mathcal{T}} v : \langle \sigma_1, \ldots, \sigma_m \rangle}{\epsilon; \Gamma \vdash_{\mathcal{T}} \iota : \sigma} \ (1 \leq i \leq m)$$

By the canonical forms lemma case (5) v is a variable x, therefore ι has the eigth form for I and it remains to show that the side conditions are satisfied. Since $\epsilon; \Gamma \vdash_{\mathcal{T}} x : \langle \sigma_1, \ldots, \sigma_m \rangle$ requires that $x \in \mathrm{dom}(\Gamma)$ and $\vdash_{\mathcal{T}} H : \Gamma$ requires that $\mathrm{dom}(H) = \mathrm{dom}(\Gamma)$. There must be a $v = \langle v_1, \ldots, v_n \rangle$ such that $H(x) = v$ or $H(x) = \mathsf{tag}(v, \tau')$. Thus the first and third side conditions are satisfied. By an inspection of the rules for $\epsilon; \Gamma \vdash_{\mathcal{T}} x : \langle \sigma_1, \ldots, \sigma_m \rangle$ it must be the case that there are $\tau_1, \ldots, \tau_n$ such that $\epsilon \vdash_{\mathcal{T}} \langle \tau_1, \ldots, \tau_n \rangle \leq \langle \sigma_1, \ldots, \sigma_m \rangle$. In S_2 there is only one rule for this judgement so it must be that $n \geq m$, so since $1 \leq i \leq m$ the second side condition is also satisfied.

$\iota = v_1\ v_2$**:** In this case ι has the ninth form for I, it remains to show the side condition is satisfied. The derivation of $\epsilon; \Gamma \vdash_{\mathcal{T}} \iota : \sigma$ must have the form:

$$\frac{\epsilon; \Gamma \vdash_{\mathcal{T}} v_1 : \sigma_1 \to \sigma_2 \quad \cdots}{\epsilon; \Gamma \vdash_{\mathcal{T}} \iota : \sigma}$$

By the canonical forms lemma case (6) v has the form $\mathsf{fix}\, f(x : \tau_1) : \tau_2.b$ as required.

$\iota = \mathsf{let}\, x_1 = v_1\, \mathsf{and}\, x_2 = v_2\, \mathsf{in}\, e'$**:** Clearly ι has the tenth form for I and there are no side conditions that need to be satisfied.

□

Theorem A.13 (Type Safety) *If* $\vdash_{\mathcal{T}} P_1 : \tau$ *and* $P_1 \mapsto^* P_2$ *then* P_2 *is not stuck.*

Proof: The proof proceeds by induction of the length of $P_1 \mapsto P_2$, the type preservation lemma, and the progress lemma. □

B Type Correctness of Translation

The main result of this appendix is that the translation preserves subtyping, typability and the types of various syntactic constructs. These theorems are interpersed with some technical lemmas, particular a typing derivation for the reified tag checker $\mathit{tagchk}(y_1, y_2, \tau, \sigma)$. These results make fairly substantial use of the weakening lemma from Appendix A, but are otherwise self contained.

Lemma B.1 $\mathrm{ftv}(\mathit{tag}(\phi, \tau)) = \mathrm{ftv}(\tau)$ *and* $\mathrm{ftv}(\mathit{tag}'(\tau)) \subseteq \mathrm{ftv}(\tau)$.

Proof: By inspection notice that $\mathrm{ftv}(\mathit{tag}(\phi, \tau)) = \mathrm{ftv}(\tau) \cup \mathrm{ftv}(\mathit{tag}'(\tau))$ and $\mathrm{ftv}(\mathit{tag}'(\tau)) = \mathrm{ftv}(\tau) - \{\alpha\}$, the desired result follows immediately. □

Theorem B.2 *For all types* τ *in the tagging language* $\epsilon \vdash_{\mathcal{T}} [\![\tau]\!]_{\mathrm{type}}$.

Proof: The desired result follows if $\mathrm{ftv}(\llbracket\tau\rrbracket_{\mathrm{type}}) = \emptyset$. The proof of the latter proceeds by induction of the structure of τ. Consider the forms that τ could take:

$\tau = \mathsf{tag}(\sigma)$**:** The desired result follows immediately from the induction hypothesis on σ and lemma B.1.

$\tau = \mathsf{tagged}$**:** By inspection $\mathrm{ftv}(\llbracket\mathsf{tagged}\rrbracket_{\mathrm{type}}) = (\mathrm{ftv}(\mathit{tag}(-,\alpha)) \cup \{\alpha\}) - \{\alpha\}$. By lemma B.1 this is equal to $(\{\alpha\} \cup \{\alpha\}) - \{\alpha\} = \emptyset$.

$\tau = \langle\tau_1, \ldots, \tau_n\rangle$**:** By inspection $\mathrm{ftv}(\llbracket\langle\tau_1, \ldots, \tau_n\rangle\rrbracket_{\mathrm{type}}) = \mathrm{ftv}(\llbracket\tau_1\rrbracket_{\mathrm{type}}) \cup \cdots \cup \mathrm{ftv}(\llbracket\tau_n\rrbracket_{\mathrm{type}})$. By the induction hypothesis this is equal to $\emptyset \cup \cdots \cup \emptyset = \emptyset$.

$\tau = \tau_1 \to \tau_2$**:** By inspection $\mathrm{ftv}(\llbracket\tau_1 \to \tau_2\rrbracket_{\mathrm{type}}) = \mathrm{ftv}(\llbracket\tau_1\rrbracket_{\mathrm{type}}) \cup \mathrm{ftv}(\llbracket\tau_2\rrbracket_{\mathrm{type}})$. By the induction hypothesis this is equal to $\emptyset \cup \emptyset = \emptyset$.

□

Theorem B.3 *If* $\vdash_{\mathcal{S}} \tau_1 \leq \tau_2$ *then* $\epsilon \vdash_{\mathcal{T}} \llbracket\tau_1\rrbracket_{\mathrm{type}} \leq \llbracket\tau_2\rrbracket_{\mathrm{type}}$.

Proof: The proof proceeds by induction on the derivation of $\vdash_{\mathcal{S}} \tau_1 \leq \tau_2$. Consider which rule was used to derive this judgement:

tag rule: The result follows by theorem B.2 and the reflexive rule in the target calculus.

tagged rule: The result follows by theorem B.2 and the reflexive rule in the target calculus.

tuple rule: By the rule it must be that $m \geq n$ and $\vdash_{\mathcal{S}} \tau_i \leq \sigma_i$. By the induction hypothesis, it must be that $\epsilon \vdash_{\mathcal{T}} \llbracket\tau_1\rrbracket_{\mathrm{type}} \leq \llbracket\sigma_i\rrbracket_{\mathrm{type}}$, so:

$$\frac{\epsilon \vdash_{\mathcal{T}} \llbracket\tau_i\rrbracket_{\mathrm{type}} \leq \llbracket\sigma_i\rrbracket_{\mathrm{type}}}{\epsilon \vdash_{\mathcal{T}} \llbracket\langle\tau_1, \ldots, \tau_n\rangle\rrbracket_{\mathrm{type}} \leq \llbracket\langle\sigma_1, \ldots, \sigma_m\rangle\rrbracket_{\mathrm{type}}}\ (m \geq n)$$

function rule: By the rule it must be that $\vdash_{\mathcal{S}} \sigma_1 \leq \tau_1$ and $\vdash_{\mathcal{S}} \tau_2 \leq \sigma_2$. By the induction hypothesis is must be that $\epsilon \vdash_{\mathcal{T}} \llbracket\sigma_1\rrbracket_{\mathrm{type}} \leq \llbracket\tau_1\rrbracket_{\mathrm{type}}$ and $\epsilon \vdash_{\mathcal{T}} \llbracket\tau_2\rrbracket_{\mathrm{type}} \leq \llbracket\sigma_2\rrbracket_{\mathrm{type}}$, so:

$$\frac{\epsilon \vdash_{\mathcal{T}} \llbracket\sigma_1\rrbracket_{\mathrm{type}} \leq \llbracket\tau_1\rrbracket_{\mathrm{type}} \quad \epsilon \vdash_{\mathcal{T}} \llbracket\tau_2\rrbracket_{\mathrm{type}} \leq \llbracket\sigma_2\rrbracket_{\mathrm{type}}}{\epsilon \vdash_{\mathcal{T}} \llbracket\tau_1 \to \sigma_1\rrbracket_{\mathrm{type}} \leq \llbracket\tau_2 \to \sigma_2\rrbracket_{\mathrm{type}}}$$

□

Lemma B.4 *If* $\Delta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2$ *and* $\phi \in \{-, \circ\}$ *then* $\Delta \vdash_{\mathcal{T}} \mathit{tag}(\phi, \tau_2) \leq \mathit{tag}(-, \tau_1)$.

Proof: First a derivation for $\Delta \vdash_{\mathcal{T}} \mathit{tag}'(\tau_2) \leq \mathit{tag}'(\tau_1)$:

$$\dfrac{\dfrac{\dfrac{\Delta, \beta, \alpha \leq \beta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2 \quad \dfrac{\Delta, \beta, \alpha \leq \beta \vdash_{\mathcal{T}} \alpha \leq \beta}{\Delta, \beta, \alpha \leq \beta \vdash_{\mathcal{T}} \langle\alpha\rangle \leq \langle\beta\rangle}}{\Delta, \beta, \alpha \leq \beta \vdash_{\mathcal{T}} \mathsf{tag}^-(\tau_2, \langle\alpha\rangle) \leq \mathsf{tag}^-(\tau_1, \langle\beta\rangle)}}{\Delta, \beta, \alpha \leq \beta \vdash_{\mathcal{T}} \mathsf{tag}^-(\tau_2, \langle\alpha\rangle)? \leq \mathsf{tag}^-(\tau_1, \langle\beta\rangle)?}}{\Delta \vdash_{\mathcal{T}} \mathit{tag}'(\tau_2) \leq \mathit{tag}'(\tau_1)}$$

Note that the weakening lemma was used to get $\Delta, \beta, \alpha \leq \beta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2$. Second a derivation for $\Delta \vdash_{\mathcal{T}} \mathit{tag}(\phi, \tau_2) \leq \mathit{tag}(-, \tau_1)$:

$$\frac{\Delta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2 \quad \dfrac{\Delta \vdash_{\mathcal{T}} \mathit{tag}'(\tau_2) \leq \mathit{tag}'(\tau_1)}{\Delta \vdash_{\mathcal{T}} \langle \mathit{tag}'(\tau_2)\rangle \leq \langle \mathit{tag}'(\tau_1)\rangle}}{\Delta \vdash_{\mathcal{T}} \mathit{tag}(\phi, \tau_2) \leq \mathit{tag}(-, \tau_1)}$$

□

Lemma B.5 *If* $\epsilon \vdash_{\mathcal{T}} \sigma$, α *is arbitrary,* $\Gamma(y_1) = \langle \mathit{tag}(-, \alpha), \alpha\rangle$, *and* $\Gamma(y_2) = \mathit{tag}(+, \sigma)$ *then* $\alpha; \Gamma \vdash_{\mathcal{T}} \mathit{tagchk}(y_1, y_2, \alpha, \sigma) : \mathit{tag}(-, \alpha) \to \sigma?$.

Proof: First, by the weakening lemma $\alpha \vdash_{\mathcal{T}} \sigma$. Second, by lemma B.1 $\mathrm{ftv}(\mathit{tag}(-, \alpha)) = \{\alpha\}$, and by target types well formedness rule $\mathrm{ftv}(\sigma) = \emptyset$. Thus $\mathrm{ftv}(\mathit{tag}(-, \alpha) \to \sigma?) = \{\alpha\}$ and $\alpha \vdash_{\mathcal{T}} \mathit{tag}(-, \alpha) \to \sigma?$. Let $\Gamma' = \Gamma, y_3 : \mathit{tag}(-, \alpha) \to \sigma?, y_4 : \mathit{tag}(-, \alpha)$. By the weakening lemma $\alpha \leq \sigma; \Gamma' \vdash_{\mathcal{T}} y_1 : \langle \mathit{tag}(-, \alpha), \alpha\rangle$ and $\alpha; \Gamma' \vdash_{\mathcal{T}} y_2 : \mathsf{tag}^{+}(\sigma, \langle \mathit{tag}'(\sigma)\rangle)$. Let the A be the following derivation:

$$\cfrac{\cfrac{\cfrac{\alpha \leq \sigma; \Gamma' \vdash_{\mathcal{T}} y_1 : \langle \mathit{tag}(-, \alpha), \alpha\rangle}{\alpha \leq \sigma; \Gamma' \vdash_{\mathcal{T}} y_1.2 : \alpha} \qquad \alpha \leq \sigma \vdash_{\mathcal{T}} \alpha \leq \sigma}{\alpha \leq \sigma; \Gamma' \vdash_{\mathcal{T}} y_1.2 : \sigma}}{\alpha \leq \sigma; \Gamma' \vdash_{\mathcal{T}} \mathsf{some}(y_1.2) : \sigma?}$$

Let B be the following derivation:

$$\frac{\alpha; \Gamma', y_5 : \mathit{tag}(-, \alpha) \vdash_{\mathcal{T}} y_3 : \mathit{tag}(-, \alpha) \to \sigma? \quad \alpha; \Gamma', y_5 : \mathit{tag}(-, \alpha) \vdash_{\mathcal{T}} y_5 : \mathit{tag}(-, \alpha)}{\alpha; \Gamma', y_5 : \mathit{tag}(-, \alpha) \vdash_{\mathcal{T}} y_3\ y_5 : \sigma?}$$

Note that the first unrolling of $\mathit{tag}'(\alpha)$ is $\mathit{tag}(-, \alpha)?$. Using this, let C be the following derivation:

$$\cfrac{\cfrac{\cfrac{\cfrac{\alpha; \Gamma' \vdash_{\mathcal{T}} y_4 : \mathit{tag}(-, \alpha)}{\alpha; \Gamma' \vdash_{\mathcal{T}} y_4 : \langle \mathit{tag}'(\alpha)\rangle}\ \text{(tag subsume)}}{\alpha; \Gamma' \vdash_{\mathcal{T}} y_4.1 : \mathit{tag}'(\alpha)}}{\alpha; \Gamma' \vdash_{\mathcal{T}} \mathsf{unroll}(y_4.1) : \mathit{tag}(-, \alpha)?} \qquad B \qquad \cfrac{\alpha \vdash_{\mathcal{T}} \sigma}{\alpha; \Gamma \vdash_{\mathcal{T}} \mathsf{none}^{\sigma} : \sigma?}}{\alpha; \Gamma' \vdash_{\mathcal{T}} \mathsf{if?}\ \mathsf{unroll}(y_4.1)\ \mathsf{then}\ y5.y3\ y_5\ \mathsf{else}\ \mathsf{none}^{\sigma}\ \mathsf{fi} : \sigma?}$$

The desired result follows from this derivation:

$$\cfrac{\alpha \vdash_{\mathcal{T}} \mathit{tag}(-, \alpha) \to \sigma? \quad \cfrac{\begin{array}{c}\epsilon \vdash_{\mathcal{T}} \sigma \\ \alpha; \Gamma' \vdash_{\mathcal{T}} y_4 : \mathsf{tag}^{-}(\alpha, \langle \mathit{tag}'(\alpha)\rangle) \\ \alpha; \Gamma' \vdash_{\mathcal{T}} y_2 : \mathsf{tag}^{+}(\sigma, \langle \mathit{tag}'(\sigma)\rangle) \\ A \\ C\end{array}}{\alpha; \Gamma' \vdash_{\mathcal{T}} \mathsf{if}\ y_4 = y_2\ \mathsf{then}\ \mathsf{some}(y_2.2)\ \mathsf{else}\ \mathsf{if?}\ \cdots\ \mathsf{then}\ \cdots\ \mathsf{else}\ \cdots\ \mathsf{fi}\ \mathsf{fi} : \sigma?}}{\alpha; \Gamma \vdash_{\mathcal{T}} \mathit{tagchk}(y_1, y_2, \alpha, \sigma) : \mathit{tag}(-, \alpha) \to \sigma?}$$

□

Theorem B.6 *If* $\Gamma \vdash_{\mathcal{S}} e : \tau$ *then* $\epsilon; [\![\Gamma]\!]_{\mathrm{ctxt}} \vdash_{\mathcal{T}} [\![e]\!]_{\mathrm{exp}} : [\![\tau]\!]_{\mathrm{type}}$.

Proof: The proof proceeds by induction on the derivation of $\Gamma \vdash_{\mathcal{S}} e : \tau$. Consider the last rule used in the derivation:

subsumption rule: In this case $\Gamma \vdash_{\mathcal{S}} e : \tau_1$ and $\vdash_{\mathcal{S}} \tau_1 \leq \tau_2$ so by the induction hypothesis and theorem B.3, $\epsilon; \llbracket \Gamma \rrbracket_{\text{ctxt}} \vdash_{\mathcal{T}} e : \llbracket \tau_1 \rrbracket_{\text{type}}$ and $\epsilon \vdash_{\mathcal{T}} \llbracket \tau_1 \rrbracket_{\text{type}} \leq \llbracket \tau_2 \rrbracket_{\text{type}}$. The desired result follows by target subsumption.

variable rule: In this case $\Gamma(x) = \tau$ so by the definition of $\llbracket \Gamma \rrbracket_{\text{ctxt}}$, $\llbracket \Gamma \rrbracket_{\text{ctxt}}(x) = \llbracket \tau \rrbracket_{\text{type}}$ and the desired result follows by the target variable rule.

newtag rule: In this case $e = \mathsf{newtag}(\sigma)$ and $\tau = \mathsf{tag}(\sigma)$. By theorem B.2 $\epsilon \vdash_{\mathcal{T}} \llbracket \sigma \rrbracket_{\text{type}}$, and by lemma B.1 $\epsilon \vdash_{\mathcal{T}} \mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}})$. Also note that one unrolling of $\mathit{tag}'(\llbracket \sigma \rrbracket_{\text{type}})$ is $\mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}})?$. The desired result follows by this derivation:

$$\dfrac{\dfrac{\dfrac{\epsilon \vdash_{\mathcal{T}} \mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}})}{\epsilon; \llbracket \Gamma \rrbracket_{\text{ctxt}} \vdash_{\mathcal{T}} \mathsf{none}^{\mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}})} : \mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}})?}}{\epsilon; \llbracket \Gamma \rrbracket_{\text{ctxt}} \vdash_{\mathcal{T}} \mathsf{roll}^{\mathit{tag}'(\llbracket \sigma \rrbracket_{\text{type}})}(\mathsf{none}^{\mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}})}) : \mathit{tag}'(\llbracket \sigma \rrbracket_{\text{type}})} \quad \epsilon \vdash_{\mathcal{T}} \llbracket \sigma \rrbracket_{\text{type}}}{\epsilon; \llbracket \Gamma \rrbracket_{\text{ctxt}} \vdash_{\mathcal{T}} \llbracket e \rrbracket_{\text{exp}} : \mathit{tag}(\circ, \llbracket \sigma \rrbracket_{\text{type}})}$$

subtag rule: In this case $e = \mathsf{subtag}(e_1, \sigma)$, $\Gamma \vdash_{\mathcal{S}} e_1 : \sigma_1$, and $\vdash_{\mathcal{S}} \sigma \leq \sigma_1$. By theorem B.3 $\epsilon \vdash_{\mathcal{T}} \llbracket \sigma \rrbracket_{\text{type}} \leq \llbracket \sigma_1 \rrbracket_{\text{type}}$, and by lemma B.4 $\epsilon \vdash_{\mathcal{T}} \mathit{tag}(\circ, \llbracket \sigma_1 \rrbracket_{\text{type}}) \leq \mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}})$. By the induction hypothesis $\epsilon; \llbracket \Gamma \rrbracket_{\text{ctxt}} \vdash_{\mathcal{T}} \llbracket e_1 \rrbracket_{\text{exp}} : \mathit{tag}(\circ, \llbracket \sigma_1 \rrbracket_{\text{type}})$, so by target subsumption $\epsilon; \llbracket \Gamma \rrbracket_{\text{ctxt}} \vdash_{\mathcal{T}} \llbracket e_1 \rrbracket_{\text{exp}} : \mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}})$. Also note that one unrolling of $\mathit{tag}'(\llbracket \sigma \rrbracket_{\text{type}})$ is $\mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}})?$. The desired result follows by this derivation:

$$\dfrac{\dfrac{\dfrac{\epsilon; \llbracket \Gamma \rrbracket_{\text{ctxt}} \vdash_{\mathcal{T}} \llbracket e_1 \rrbracket_{\text{exp}} : \mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}})}{\epsilon; \llbracket \Gamma \rrbracket_{\text{ctxt}} \vdash_{\mathcal{T}} \mathsf{some}(\llbracket e_1 \rrbracket_{\text{exp}}) : \mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}})?}}{\epsilon; \llbracket \Gamma \rrbracket_{\text{ctxt}} \vdash_{\mathcal{T}} \mathsf{roll}^{\mathit{tag}'(\llbracket \sigma \rrbracket_{\text{type}})}(\mathsf{some}(\llbracket e_1 \rrbracket_{\text{exp}})) : \mathit{tag}'(\llbracket \sigma \rrbracket_{\text{type}})} \quad \epsilon \vdash_{\mathcal{T}} \llbracket \sigma \rrbracket_{\text{type}}}{\epsilon; \llbracket \Gamma \rrbracket_{\text{ctxt}} \vdash_{\mathcal{T}} \llbracket e \rrbracket_{\text{exp}} : \mathit{tag}(\circ, \llbracket \sigma \rrbracket_{\text{type}})}$$

tagged rule: In this case $e = \mathsf{tagged}(e')$, $\Gamma \vdash_{\mathcal{S}} e' : \langle \mathsf{tag}(\sigma), \sigma \rangle$, and $\tau = \mathsf{tagged}$. By the induction hypothesis $\epsilon; \llbracket \Gamma \rrbracket_{\text{ctxt}} \vdash_{\mathcal{T}} e' : \langle \mathit{tag}(\circ, \llbracket \sigma \rrbracket_{\text{type}}), \llbracket \sigma \rrbracket_{\text{type}} \rangle$. The following is a subtyping derivation:

$$\dfrac{\dfrac{\epsilon \vdash_{\mathcal{T}} \llbracket \sigma \rrbracket_{\text{type}} \leq \llbracket \sigma \rrbracket_{\text{type}} \quad \epsilon \vdash_{\mathcal{T}} \langle \mathit{tag}'(\llbracket \sigma \rrbracket_{\text{type}}) \rangle \leq \langle \mathit{tag}'(\llbracket \sigma \rrbracket_{\text{type}}) \rangle}{\epsilon \vdash_{\mathcal{T}} \mathit{tag}(\circ, \llbracket \sigma \rrbracket_{\text{type}}) \leq \mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}})} \quad \epsilon \vdash_{\mathcal{T}} \llbracket \sigma \rrbracket_{\text{type}} \leq \llbracket \sigma \rrbracket_{\text{type}}}{\epsilon \vdash_{\mathcal{T}} \langle \mathit{tag}(\circ, \llbracket \sigma \rrbracket_{\text{type}}), \llbracket \sigma \rrbracket_{\text{type}} \rangle \leq \langle \mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}}), \llbracket \sigma \rrbracket_{\text{type}} \rangle}$$

So by subsumption $\epsilon; \llbracket \Gamma \rrbracket_{\text{ctxt}} \vdash_{\mathcal{T}} e' : \langle \mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}}), \llbracket \sigma \rrbracket_{\text{type}} \rangle$. The dersired result then follows by the following derivation, which uses the pack rule:

$$\dfrac{\epsilon \vdash_{\mathcal{T}} \llbracket \sigma \rrbracket_{\text{type}} \quad \epsilon; \llbracket \Gamma \rrbracket_{\text{ctxt}} \vdash_{\mathcal{T}} \llbracket e' \rrbracket_{\text{exp}} : \langle \mathit{tag}(-, \llbracket \sigma \rrbracket_{\text{type}}), \llbracket \sigma \rrbracket_{\text{type}} \rangle}{\epsilon; \llbracket \Gamma \rrbracket_{\text{ctxt}} \vdash_{\mathcal{T}} \llbracket e \rrbracket_{\text{exp}} : \llbracket \mathsf{tagged} \rrbracket_{\text{type}}}$$

if tag rule: In this case $e = \mathsf{iftagof}\, e_1 = e_2\, \mathsf{then}\, x.b_1\, \mathsf{else}\, b_2\, \mathsf{fi}$, $\Gamma \vdash_{\mathcal{S}} e_1 : \mathsf{tagged}$, $\Gamma \vdash_{\mathcal{S}} e_2 : \mathsf{tag}(\sigma)$, $\Gamma, x : \sigma \vdash_{\mathcal{S}} b_1 : \tau$, and $\Gamma \vdash_{\mathcal{S}} b_2 : \tau$. Let Γ' be:

$$\llbracket \Gamma \rrbracket_{\text{ctxt}}, x_1 : \llbracket \mathsf{tagged} \rrbracket_{\text{type}}, x_2 : \mathit{tag}(+, \llbracket \sigma \rrbracket_{\text{type}}), y_1 : \langle \mathit{tag}(-, \alpha), \alpha \rangle$$

By theorem B.2 $\epsilon \vdash_{\mathcal{T}} [\![\sigma]\!]_{\text{type}}$, so by lemma B.5:

$$\alpha; \Gamma' \vdash_{\mathcal{T}} \mathit{tagchk}(y_1, y_2, \alpha, [\![\sigma]\!]_{\text{type}}) : \mathit{tag}(-, \alpha) \to [\![\sigma]\!]_{\text{type}}?$$

Let A be the following derivation:

$$\frac{\alpha; \Gamma' \vdash_{\mathcal{T}} \mathit{tagchk}(y_1, x_2, \alpha, [\![\sigma]\!]_{\text{type}}) : \mathit{tag}(-, \alpha) \to [\![\sigma]\!]_{\text{type}}? \quad \dfrac{\alpha; \Gamma' \vdash_{\mathcal{T}} y_1 : \langle \mathit{tag}(-, \alpha), \alpha \rangle}{\alpha; \Gamma' \vdash_{\mathcal{T}} y_1.1 : \mathit{tag}(-, \alpha)}}{\alpha; \Gamma' \vdash_{\mathcal{T}} \mathit{tagchk}(y_1, x_2, \alpha, [\![\sigma]\!]_{\text{type}})\ y_1.1 : [\![\sigma]\!]_{\text{type}}?}$$

By the induction hypothesis $\epsilon; [\![\Gamma, x : \sigma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![b_1]\!]_{\text{exp}} : [\![\tau]\!]_{\text{type}}$ and $\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![b_2]\!]_{\text{exp}} : [\![\tau]\!]_{\text{type}}$. By the weakening lemma $\alpha; \Gamma', x : [\![\sigma]\!]_{\text{type}} \vdash_{\mathcal{T}} [\![b_1]\!]_{\text{exp}} : [\![\tau]\!]_{\text{type}}$ and $\alpha; \Gamma' \vdash_{\mathcal{T}} [\![b_2]\!]_{\text{exp}} : [\![\tau]\!]_{\text{type}}$. Let B be the following derivation:

$$\frac{A \quad \alpha; \Gamma', x : [\![\sigma]\!]_{\text{type}} \vdash_{\mathcal{T}} [\![b_1]\!]_{\text{exp}} : [\![\tau]\!]_{\text{type}} \quad \alpha; \Gamma' \vdash_{\mathcal{T}} [\![b_2]\!]_{\text{exp}} : [\![\tau]\!]_{\text{type}}}{\alpha; \Gamma' \vdash_{\mathcal{T}} \mathsf{if?}\ \mathit{tagchk}(y_1, x_2, \alpha, [\![\sigma]\!]_{\text{type}})\ y_1.1\ \mathsf{then}\ x.[\![b_1]\!]_{\text{exp}}\ \mathsf{else}\ [\![b_2]\!]_{\text{exp}}\ \mathsf{fi} : [\![\tau]\!]_{\text{type}}}$$

By theorem B.2 $\epsilon \vdash_{\mathcal{T}} [\![\tau]\!]_{\text{type}}$. Let C be the following derivation:

$$\frac{\epsilon; [\![\Gamma]\!]_{\text{ctxt}}, x_1{:}[\![\mathsf{tagged}]\!]_{\text{type}}, x_2{:}\mathit{tag}(+, [\![\sigma]\!]_{\text{type}}) \vdash_{\mathcal{T}} x_1 : \exists\alpha.\langle \mathit{tag}(-, \alpha), \alpha \rangle \quad B \quad \epsilon \vdash_{\mathcal{T}} [\![\tau]\!]_{\text{type}}}{\epsilon; [\![\Gamma]\!]_{\text{ctxt}}, x_1{:}[\![\mathsf{tagged}]\!]_{\text{type}}, x_2{:}\mathit{tag}(+, [\![\sigma]\!]_{\text{type}}) \vdash_{\mathcal{T}} \mathsf{unpack}[\alpha, y_1] = x_1\ \mathsf{in}\ \cdots : [\![\tau]\!]_{\text{type}}}$$

Since $\epsilon \vdash_{\mathcal{T}} [\![\tau]\!]_{\text{type}}$ it must be that $\text{ftv}([\![\tau]\!]_{\text{type}}) = \emptyset$ so by lemma B.1 $\text{ftv}(\mathit{tag}'([\![\tau]\!]_{\text{type}})) = \emptyset$ so $\text{ftv}(\langle \mathit{tag}'([\![\tau]\!]_{\text{type}}) \rangle) = \emptyset$ so $\epsilon \vdash_{\mathcal{T}} \langle \mathit{tag}'([\![\tau]\!]_{\text{type}}) \rangle$. Thus $\epsilon \vdash_{\mathcal{T}} [\![\tau]\!]_{\text{type}} \leq [\![\tau]\!]_{\text{type}}$ and $\epsilon \vdash_{\mathcal{T}} \langle \mathit{tag}'([\![\tau]\!]_{\text{type}}) \rangle \leq \langle \mathit{tag}'([\![\tau]\!]_{\text{type}}) \rangle$. By the tag subtyping rule $\epsilon \vdash_{\mathcal{T}} \mathit{tag}(\circ, [\![\sigma]\!]_{\text{type}}) \leq \mathit{tag}(+, [\![\sigma]\!]_{\text{type}})$. By the induction hypothesis $\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![e_2]\!]_{\text{exp}} : \mathit{tag}(\circ, [\![\sigma]\!]_{\text{type}})$ So let D be the following derivation:

$$\frac{\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![e_2]\!]_{\text{exp}} : \mathit{tag}(\circ, [\![\sigma]\!]_{\text{type}}) \quad \epsilon \vdash_{\mathcal{T}} \mathit{tag}(\circ, [\![\sigma]\!]_{\text{type}}) \leq \mathit{tag}(+, [\![\sigma]\!]_{\text{type}})}{\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![e_2]\!]_{\text{exp}} : \mathit{tag}(+, [\![\sigma]\!]_{\text{type}})}$$

By the induction hypothesis $\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![e_1]\!]_{\text{exp}} : [\![\mathsf{tagged}]\!]_{\text{type}}$. The desired result follows from this derivation:

$$\frac{\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![e_1]\!]_{\text{exp}} : [\![\mathsf{tagged}]\!]_{\text{type}} \quad D \quad C}{\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![e]\!]_{\text{exp}} : [\![\tau]\!]_{\text{type}}}$$

tuple rule: In this case $e = \langle e_1, \ldots, e_n \rangle$, $\tau = \langle \tau_1, \ldots, \tau_n \rangle$, and $\Gamma \vdash_{\mathcal{S}} e_i : \tau_i$. By the induction hypothesis $\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![e_i]\!]_{\text{exp}} : [\![\tau_i]\!]_{\text{type}}$. The desired result follows by applying the target tuple rule.

projection rule: In this case $e = e'.i$, $\Gamma \vdash_{\mathcal{S}} e' : \langle \tau_1, \ldots, \tau_n \rangle$, $1 \leq i \leq n$, and $\tau = \tau_i$. By the induction hypothesis $\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![e']\!]_{\text{exp}} : [\![\langle \tau_1, \ldots, \tau_n \rangle]\!]_{\text{type}}$. The later is equal to $\langle [\![\tau_1]\!]_{\text{type}}, \ldots, [\![\tau_n]\!]_{\text{type}} \rangle$ so the desired result follows by applying the target projection rule.

function rule: In this case $e = \mathsf{fix}\ f(x{:}\tau_1){:}\tau_2.b$, $\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash_{\mathcal{S}} b : \tau_2$, and $\tau = \tau_1 \to \tau_2$. By theorem B.2 it must be that $\epsilon \vdash_{\mathcal{T}} [\![\tau_1]\!]_{\text{type}} \to [\![\tau_2]\!]_{\text{type}}$. By the induction hypothesis, it must be that $\epsilon; [\![\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![b]\!]_{\text{exp}} : [\![\tau_2]\!]_{\text{type}}$. Since $[\![\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1]\!]_{\text{ctxt}} = [\![\Gamma]\!]_{\text{ctxt}}, f{:}[\![\tau_1]\!]_{\text{type}} \to [\![\tau_2]\!]_{\text{type}}, x{:}[\![\tau_1]\!]_{\text{type}}$ and $[\![\tau_1 \to \tau_2]\!]_{\text{type}} = [\![\tau_1]\!]_{\text{type}} \to [\![\tau_2]\!]_{\text{type}}$, the desired result follows by an application of the fix rule.

application rule: In this case $e = e_1\ e_2$, $\Gamma \vdash_{\mathcal{S}} e_1 : \tau_2 \to \tau$, and $\Gamma \vdash_{\mathcal{S}} e_2 : \tau_2$. By the induction hypothesis $\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![e_1]\!]_{\text{exp}} : [\![\tau_2 \to \tau]\!]_{\text{type}}$ and $\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![e_2]\!]_{\text{exp}} : [\![\tau_2]\!]_{\text{type}}$. Since $[\![\tau_2 \to \tau]\!]_{\text{type}} = [\![\tau_2]\!]_{\text{type}} \to [\![\tau]\!]_{\text{type}}$, the desired result follows by applying the target application rule.

□

Theorem B.7 *If* $\Gamma \vdash_{\mathcal{S}} h : \tau$ *then* $[\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![h]\!]_{\text{hval}} : [\![\tau]\!]_{\text{type}}$.

Proof: There are three cases:

$h = (\sigma, \epsilon)$**:** In this case $\tau = \mathsf{tag}(\sigma)$. By theorem B.2 $\epsilon \vdash_{\mathcal{T}} [\![\sigma]\!]_{\text{type}}$, and by lemma B.1 $\epsilon \vdash_{\mathcal{T}} \mathit{tag}(-, [\![\sigma]\!]_{\text{type}})$. Also note that one unrolling of $\mathit{tag}'([\![\sigma]\!]_{\text{type}})$ is $\mathit{tag}(-, [\![\sigma]\!]_{\text{type}})?$. The desired result follows by this derivation:

$$\dfrac{\dfrac{\dfrac{\epsilon \vdash_{\mathcal{T}} \mathit{tag}(-, [\![\sigma]\!]_{\text{type}})}{\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} \mathsf{none}^{\mathit{tag}(-, [\![\sigma]\!]_{\text{type}})} : \mathit{tag}(-, [\![\sigma]\!]_{\text{type}})?}}{\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} \mathsf{roll}^{\mathit{tag}'([\![\sigma]\!]_{\text{type}})}(\mathsf{none}^{\mathit{tag}(-, [\![\sigma]\!]_{\text{type}})}) : \mathit{tag}'([\![\sigma]\!]_{\text{type}})} \quad \epsilon \vdash_{\mathcal{T}} [\![\sigma]\!]_{\text{type}}}{\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![h]\!]_{\text{hval}} : \mathit{tag}(\circ, [\![\sigma]\!]_{\text{type}})}$$

$h = (\sigma, x)$**:** In this case $\Gamma \vdash_{\mathcal{S}} x : \mathsf{tag}(\sigma_x)$, $\vdash_{\mathcal{S}} \sigma \leq \sigma_x$, and $\tau = \mathsf{tag}(\sigma)$. By theorem B.3 $\epsilon \vdash_{\mathcal{T}} [\![\sigma]\!]_{\text{type}} \leq [\![\sigma_x]\!]_{\text{type}}$, and by lemma B.4 $\epsilon \vdash_{\mathcal{T}} \mathit{tag}(\circ, [\![\sigma_x]\!]_{\text{type}}) \leq \mathit{tag}(-, [\![\sigma]\!]_{\text{type}})$. By theorem B.6 $\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} x : \mathit{tag}(\circ, [\![\sigma_x]\!]_{\text{type}})$, so by target subsumption $\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} x : \mathit{tag}(-, [\![\sigma]\!]_{\text{type}})$. Also note that one unrolling of $\mathit{tag}'([\![\sigma]\!]_{\text{type}})$ is $\mathit{tag}(-, [\![\sigma]\!]_{\text{type}})?$. The desired result follows by this derivation:

$$\dfrac{\dfrac{\dfrac{\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} x : \mathit{tag}(-, [\![\sigma]\!]_{\text{type}})}{\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} \mathsf{some}(x) : \mathit{tag}(-, [\![\sigma]\!]_{\text{type}})?}}{\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} \mathsf{roll}^{\mathit{tag}'([\![\sigma]\!]_{\text{type}})}(\mathsf{some}(x)) : \mathit{tag}'([\![\sigma]\!]_{\text{type}})} \quad \epsilon \vdash_{\mathcal{T}} [\![\sigma]\!]_{\text{type}}}{\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![h]\!]_{\text{hval}} : \mathit{tag}(\circ, [\![\sigma]\!]_{\text{type}})}$$

$h = \langle v_1, \ldots, v_n \rangle$**:** In this case $\tau = \langle \tau_1, \ldots, \tau_n \rangle$ and $\Gamma \vdash_{\mathcal{S}} v_i : \tau_i$. By theorem B.6 $\epsilon; [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![v_i]\!]_{\text{exp}} : [\![\tau_i]\!]_{\text{type}}$. The desired result follows by applying the target tuple rule.

□

Theorem B.8 *If* $\vdash_{\mathcal{S}} H : \Gamma$ *then* $\vdash_{\mathcal{T}} [\![H]\!]_{\text{heap}} : [\![\Gamma]\!]_{\text{ctxt}}$.

Proof: Let $H = x_1 = h_1, \ldots, x_n = h_n$ and $\Gamma = x_1{:}\tau_1, \ldots, x_n{:}\tau_n$, then it must be that $x_1{:}\tau_1, \ldots, x_{i-1}{:}\tau_{i-1} \vdash_{\mathcal{S}} h_i : \tau_i$. By theorem B.7 $[\![x_1{:}\tau_1, \ldots, x_{i-1}{:}\tau_{i-1}]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![h_i]\!]_{\text{hval}} : [\![\tau_i]\!]_{\text{type}}$. Now $[\![x_1{:}\tau_1, \ldots, x_{i-1}{:}\tau_{i-1}]\!]_{\text{ctxt}} = x_1{:}[\![\tau_1]\!]_{\text{type}}, \ldots, x_{i-1}{:}[\![\tau_{i-1}]\!]_{\text{type}}$. So the desired result follows by applying the target heap rule. □

Theorem B.9 *If* $\vdash_{\mathcal{S}} P : \tau$ *then* $\vdash_{\mathcal{T}} [\![P]\!]_{\text{prog}} : [\![\tau]\!]_{\text{type}}$.

Proof: Let $P = \mathsf{let}\, H\, \mathsf{in}\, e$. There must be a Γ such that $\vdash_{\mathcal{S}} H : \Gamma$ and $\Gamma \vdash_{\mathcal{S}} e : \tau$. By theorem B.8 $\vdash_{\mathcal{T}} [\![H]\!]_{\text{heap}} : [\![\Gamma]\!]_{\text{ctxt}}$ and by theorem B.6 $[\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{T}} [\![e]\!]_{\text{exp}} : [\![\tau]\!]_{\text{type}}$. The desired result follows by the target program rule. □

C Operational Correctness

The purpose of this appendix is to show that the translation preserves the operational semantics of tagging language in the target language. In particular I show that the translated term simulates the source term, which is usually considered a sufficient way to show semantics preservation.

Define $[\![E]\!]_{\text{ectxt}}$ to be $[\![E[x]]\!]_{\text{exp}}[x := []]$ for some fresh x. By a straighforward induction it follows that $[\![E]\!]_{\text{ectxt}}$ is a target evaluation context.

Lemma C.1 $[\![e_1[x := e_2]]\!]_{\text{exp}} = [\![e_1]\!]_{\text{exp}}[x := [\![e_2]\!]_{\text{exp}}]$

Proof: The proof proceeds by induciton of the structure of e_1 and reduces in a straight forward manner to the induction hypothesis applied to the subterms of e_1. □

Lemma C.2 $[\![E[e]]\!]_{\text{exp}} = [\![E]\!]_{\text{ectxt}}[[\![e]\!]_{\text{exp}}]$

Proof: The result is immediate from the definition of $[\![\cdot]\!]_{\text{ectxt}}$ and lemma C.1 and the properties of substitution.
□

Lemma C.3 *If* $\vdash_{\mathcal{S}} H : \Gamma$, $\Gamma \vdash_{\mathcal{S}} x : \mathsf{tag}(\tau_x)$, *and* y *is arbitrary then there exists an* $n \geq 1$ *such that there exist* $w_1, \ldots, w_n$, *and* $\tau_1, \ldots, \tau_n$, *such that* $y \notin \{w_1, \ldots, w_{n-1}\}$, $x = w_1$, $\forall 1 \leq i < n : H(w_i) = (\tau_i, w_{i+1})$, *and:*

$$\begin{array}{ll} & \mathit{tagchk}^H(x, y) \wedge w_n = y \\ \vee & \mathit{not}\ \mathit{tagchk}^H(x, y) \wedge w_n \neq y \wedge H(w_n) = (\tau_n, \epsilon) \end{array}$$

Proof: The proof proceeds by induction on the size of H. If $x = y$ then take $n = 1$, $w_1 = y$, and $\tau_1 = \tau_x$ and the result clearly follows. Otherwise, by an inspection of the tagging language typing rules and induction of the derivation of $H \vdash_{\mathcal{S}} \Gamma :$ it must be that $H = H_1, x = h_x, H_2$, $\vdash_{\mathcal{S}} H_1 : \Gamma_1$, and $\Gamma_1 \vdash_{\mathcal{S}} h_x : \mathsf{tag}(\tau_x)$. By inspection of the tagging languagetyping rules there are two possible cases for h_x: (τ_x, ϵ) and (τ_x, w), so consider these cases:

$h_x = (\tau_x, \epsilon)$**:** In this case $\mathit{tagchk}^H(x, y)$ does not hold. The required result follows if $n = 1$, $w_1 = x$, and $\tau_1 = \tau_x$.

$h_x = (\tau_x, w)$**:** By the tagging language typing rules $\Gamma_1 \vdash_{\mathcal{S}} w : \mathsf{tag}(\tau_w)$ for some τ_w. Since H_1 is smaller than H, by the induction hypothesis there exists $m \geq 1$, $z_1, \ldots, z_m$, and $\sigma_1, \ldots, \sigma_m$ such that $y \notin \{z_1, \ldots, z_{m-1}\}$, $w = z_1$, $\forall 1 \leq i < m : H(z_i) = (\sigma_i, z_{i+1})$, and:

$$\begin{array}{ll} & \mathit{tagchk}^H(x, y) \wedge w_n = y \\ \vee & \text{not } \mathit{tagchk}^H(x, y) \wedge w_n \neq y \wedge H(w_n) = (\tau_n, \epsilon) \end{array}$$

The required result follows if $n = m + 1$, $w_1 = x$, $w_{i+1} = z_i$ for $1 \leq i \leq m$, $\tau_1 = \tau_x$, and $\tau_{i+1} = \sigma_i$ for $1 \leq i \leq m$ and by noting that $\mathit{tagchk}^H(x, y) = \mathit{tagchk}^H(w, y)$.

□

Lemma C.4 *If* $\vdash_{\mathcal{S}} H : \Gamma$, $H(x) = \langle x', v, \vec{v}\rangle$, *and* $\Gamma \vdash_{\mathcal{S}} x' : \mathsf{tag}(\tau_1)$ *then:*

$$\mathsf{let}\ [\![H]\!]_{\mathrm{heap}}\ \mathsf{in}\ E[\mathit{tagchk}(x, y, [\![\tau_1]\!]_{\mathrm{type}}, [\![\sigma]\!]_{\mathrm{type}})\ x'] \mapsto^{+} \mathsf{let}\ [\![H]\!]_{\mathrm{heap}}\ \mathsf{in}\ E[e]$$
$$\textit{where } e = \begin{cases} \mathsf{some}(v) & \mathit{tagchk}^{H}(x', y) \\ \mathsf{none}^{[\![\sigma]\!]_{\mathrm{type}}} & \textit{not } \mathit{tagchk}^{H}(x', y) \end{cases}$$

Proof: Let $P(x) = \mathsf{let}\ [\![H]\!]_{\mathrm{heap}}\ \mathsf{in}\ [\![E]\!]_{\mathrm{ectxt}}[x]$. By lemma C.3 there exists $n \geq 1$, $w_1, \ldots, w_n$, and $\tau_1, \ldots, \tau_n$ such that $y \notin \{w_1, \ldots, w_{n-1}\}$, $x' = w_1$, $\forall 1 \leq i < n : H(w_i) = (\tau_i, w_{i+1})$ and:

$$\begin{array}{ll} & \mathit{tagchk}^{H}(x', y) \wedge w_n = y \\ \vee & \text{not } \mathit{tagchk}^{H}(x', y) \wedge w_n \neq y \wedge H(w_n) = (\tau_n, \epsilon) \end{array}$$

First I will show that for $1 \leq i < n$:

$$P(\mathit{tagchk}(x, y, [\![\tau_1]\!]_{\mathrm{type}}, [\![\sigma]\!]_{\mathrm{type}})\ w_i) \mapsto^{+} P(\mathit{tagchk}(x, y, [\![\tau_1]\!]_{\mathrm{type}}, [\![\sigma]\!]_{\mathrm{type}})\ w_{i+1})$$

Note that $w_i \neq y$ and that:

$$[\![H]\!]_{\mathrm{heap}}(w_i) = [\![(\tau_i, w_{i+1})]\!]_{\mathrm{hval}} = \mathsf{tag}(\langle \mathsf{roll}^{\mathit{tag}'([\![\tau_i]\!]_{\mathrm{type}})}(\mathsf{some}(w_{i+1}))\rangle, [\![\tau_1]\!]_{\mathrm{type}})$$

Then:

$$\begin{array}{ll} & P(\mathit{tagchk}(x, y, [\![\tau_1]\!]_{\mathrm{type}}, [\![\sigma]\!]_{\mathrm{type}})\ w_i) \\ \mapsto & P(\mathsf{if}\ w_i = y\ \mathsf{then}\ \cdots\ \mathsf{else}\ \mathsf{if?}\ \mathsf{unroll}(w_i.1)\ \mathsf{then}\ \cdots\ \mathsf{else}\ \cdots\ \mathsf{fi}\ \mathsf{fi}) \\ \mapsto & P(\mathsf{if?}\ \mathsf{unroll}(w_i.1)\ \mathsf{then}\ y_5.\mathit{tagchk}(x, y, [\![\tau_1]\!]_{\mathrm{type}}, [\![\sigma]\!]_{\mathrm{type}})\ y_5\ \mathsf{else}\ \cdots\ \mathsf{fi}) \\ \mapsto & P(\mathsf{if?}\ \mathsf{unroll}(\mathsf{roll}^{\mathit{tag}'([\![\tau_i]\!]_{\mathrm{type}})}(\mathsf{some}(w_{i+1})))\ \mathsf{then}\ y_5.\mathit{tagchk}(x, y, [\![\tau_1]\!]_{\mathrm{type}}, [\![\sigma]\!]_{\mathrm{type}})\ y_5 \\ & \quad \mathsf{else}\ \cdots\ \mathsf{fi}) \\ \mapsto & P(\mathsf{if?}\ \mathsf{some}(w_{i+1})\ \mathsf{then}\ y_5.\mathit{tagchk}(x, y, [\![\tau_1]\!]_{\mathrm{type}}, [\![\sigma]\!]_{\mathrm{type}})\ y_5\ \mathsf{else}\ \cdots\ \mathsf{fi}) \\ \mapsto & P(\mathit{tagchk}(x, y, [\![\tau_1]\!]_{\mathrm{type}}, [\![\sigma]\!]_{\mathrm{type}})\ w_{i+1}) \end{array}$$

So by induction:

$$P(\mathit{tagchk}(x, y, [\![\tau_1]\!]_{\mathrm{type}}, [\![\sigma]\!]_{\mathrm{type}})\ x') \mapsto^{*} P(\mathit{tagchk}(x, y, [\![\tau_1]\!]_{\mathrm{type}}, [\![\sigma]\!]_{\mathrm{type}})\ w_n)$$

Now there are two cases: $\mathit{tagchk}^{H}(x', y)$ or not. Assume $\mathit{tagchk}^{H}(x', y)$, then $w_n = y$ so:

$$\begin{array}{ll} & P(\mathit{tagchk}(x, y, [\![\tau_1]\!]_{\mathrm{type}}, [\![\sigma]\!]_{\mathrm{type}})\ w_n) \\ \mapsto & P(\mathsf{if}\ w_n = y\ \mathsf{then}\ \mathsf{some}(x.2)\ \mathsf{else}\ \cdots\ \mathsf{fi}) \\ \mapsto & P(\mathsf{some}(x.2)) \\ \mapsto & P(\mathsf{some}(v)) \end{array}$$

Now assume not $\mathit{tagchk}^{H}(x', y)$, then $w_n \neq y$ and $H(w_n) = (\tau_n, \epsilon)$. Note that:

$$[\![H]\!]_{\mathrm{heap}}(w_n) = [\![(\tau_n, \epsilon)]\!]_{\mathrm{hval}} = \mathsf{tag}(\langle \mathsf{roll}^{\mathit{tag}'([\![\tau_n]\!]_{\mathrm{type}})}(\mathsf{none}^{\mathit{tag}(-, [\![\tau_n]\!]_{\mathrm{type}})})\rangle, [\![\tau_n]\!]_{\mathrm{type}})$$

So:

$$\begin{array}{ll} & P(\mathit{tagchk}(x, y, [\![\tau_1]\!]_{\mathrm{type}}, [\![\sigma]\!]_{\mathrm{type}})\ w_n) \\ \mapsto & P(\mathsf{if}\ w_n = y\ \mathsf{then}\ \cdots\ \mathsf{else}\ \mathsf{if?}\ \mathsf{unroll}(w_n.1)\ \mathsf{then}\ \cdots\ \mathsf{else}\ \mathsf{none}^{[\![\sigma]\!]_{\mathrm{type}}}\ \mathsf{fi}\ \mathsf{fi}) \\ \mapsto & P(\mathsf{if?}\ \mathsf{unroll}(w_n.1)\ \mathsf{then}\ \cdots\ \mathsf{else}\ \mathsf{none}^{[\![\sigma]\!]_{\mathrm{type}}}\ \mathsf{fi}) \\ \mapsto & P(\mathsf{if?}\ \mathsf{unroll}(\mathsf{roll}^{\mathit{tag}'([\![\tau_n]\!]_{\mathrm{type}})}(\mathsf{none}^{\mathit{tag}(-, [\![\tau_n]\!]_{\mathrm{type}})}))\ \mathsf{then}\ \cdots\ \mathsf{else}\ \mathsf{none}^{[\![\sigma]\!]_{\mathrm{type}}}\ \mathsf{fi}) \\ \mapsto & P(\mathsf{if?}\ \mathsf{none}^{\mathit{tag}(-, [\![\tau_n]\!]_{\mathrm{type}})}\ \mathsf{then}\ \cdots\ \mathsf{else}\ \mathsf{none}^{[\![\sigma]\!]_{\mathrm{type}}}\ \mathsf{fi}) \\ \mapsto & P(\mathsf{none}^{[\![\sigma]\!]_{\mathrm{type}}}) \end{array}$$

$\square$

Theorem C.5 *If* $\vdash_{\mathcal{S}} P_1 : \tau$ *and* $P_1 \mapsto P_2$ *then* $[\![P_1]\!]_{\text{prog}} \mapsto^+ [\![P_2]\!]_{\text{prog}}$.

Proof: Let $P_1 = \mathsf{let}\ H_1\ \mathsf{in}\ E[I]$ and $P_2 = \mathsf{let}\ H_2\ \mathsf{in}\ E[e]$ be given by the rules in Figure 2. By the definition of $[\![\cdot]\!]_{\text{prog}}$ and lemma C.2 $[\![P_1]\!]_{\text{prog}} = \mathsf{let}\ [\![H_1]\!]_{\text{heap}}\ \mathsf{in}\ [\![E]\!]_{\text{ectxt}}[[\![I]\!]_{\text{exp}}]$ and $[\![P_2]\!]_{\text{prog}} = \mathsf{let}\ [\![H_2]\!]_{\text{heap}}\ \mathsf{in}\ [\![E]\!]_{\text{ectxt}}[[\![e]\!]_{\text{exp}}]$ so it remains to show that the former reduces to the latter is some finite nonzero number of steps. Consider the various cases for I:

$I = \mathsf{newtag}(\tau)$**:** In this case $e = x$ and $H_2 = H_1, x = (\tau, \epsilon)$. Note that $[\![I]\!]_{\text{exp}} = \mathsf{tag}(\langle\cdots\rangle, \cdots) = [\![(\tau, \epsilon)]\!]_{\text{hval}}$ is a heap value so:

$$\begin{array}{rl} & \mathsf{let}\ [\![H_1]\!]_{\text{heap}}\ \mathsf{in}\ [\![E]\!]_{\text{ectxt}}[[\![I]\!]_{\text{exp}}] \\ \mapsto & \mathsf{let}\ [\![H_1]\!]_{\text{heap}}, x = [\![(\tau, \epsilon)]\!]_{\text{hval}}\ \mathsf{in}\ [\![E]\!]_{\text{ectxt}}[x] \\ = & \mathsf{let}\ [\![H_2]\!]_{\text{heap}}\ \mathsf{in}\ [\![E]\!]_{\text{ectxt}}[[\![e]\!]_{\text{exp}}] \end{array}$$

$I = \mathsf{subtag}(y, \tau)$**:** The same argument as the previous case applies except that $[\![I]\!]_{\text{exp}}$ is a different heap value.

$I = \langle v_1, \ldots, v_n\rangle$**:** The same argument as the previous case applies except that $[\![I]\!]_{\text{exp}}$ is a different heap value.

$I = x.i$**:** In this case $e = v_i$, $H_2 = H_1$, $H_1(x) = \langle v_1, \ldots, v_n\rangle$, and $1 \leq i \leq n$. Note that $[\![I]\!]_{\text{exp}} = x.i$ and $[\![H_1]\!]_{\text{heap}}(x) = [\![H_1(x)]\!]_{\text{hval}} = \langle[\![v_1]\!]_{\text{exp}}, \ldots, [\![v_n]\!]_{\text{exp}}\rangle$. So:

$$\begin{array}{rl} & \mathsf{let}\ [\![H_1]\!]_{\text{heap}}\ \mathsf{in}\ [\![E]\!]_{\text{ectxt}}[[\![I]\!]_{\text{exp}}] \\ \mapsto & \mathsf{let}\ [\![H_1]\!]_{\text{heap}}\ \mathsf{in}\ [\![E]\!]_{\text{ectxt}}[[\![v_i]\!]_{\text{exp}}] \\ = & \mathsf{let}\ [\![H_2]\!]_{\text{heap}}\ \mathsf{in}\ [\![E]\!]_{\text{ectxt}}[[\![e]\!]_{\text{exp}}] \end{array}$$

$I = v_1\ v_2$ **and** $v_1 = \mathsf{fix}\ f(x{:}\tau_1){:}\tau_2.b$**:** In this case $e = b[f, x := v_1, v_2]$ and $H_2 = H_1$. Note that $[\![I]\!]_{\text{exp}} = [\![v_1]\!]_{\text{exp}}\ [\![v_2]\!]_{\text{exp}}$ and $[\![v_1]\!]_{\text{exp}} = \mathsf{fix}\ f(x{:}[\![\tau_1]\!]_{\text{type}}){:}[\![\tau_2]\!]_{\text{type}}.[\![b]\!]_{\text{exp}}$. So:

$$\begin{array}{rl} & \mathsf{let}\ [\![H_1]\!]_{\text{heap}}\ \mathsf{in}\ [\![E]\!]_{\text{ectxt}}[[\![I]\!]_{\text{exp}}] \\ \mapsto & \mathsf{let}\ [\![H_1]\!]_{\text{heap}}\ \mathsf{in}\ [\![E]\!]_{\text{ectxt}}[[\![b]\!]_{\text{exp}}[f, x := [\![v_1]\!]_{\text{exp}}, [\![v_2]\!]_{\text{exp}}]] \\ = & \mathsf{let}\ [\![H_1]\!]_{\text{heap}}\ \mathsf{in}\ [\![E]\!]_{\text{ectxt}}[[\![b[f, x := v_1, v_2]]\!]_{\text{exp}}] \\ = & \mathsf{let}\ [\![H_2]\!]_{\text{heap}}\ \mathsf{in}\ [\![E]\!]_{\text{ectxt}}[[\![e]\!]_{\text{exp}}] \end{array}$$

Note that lemma C.1 was used twice in the second step.

$I = \mathsf{iftagof}\ \mathsf{tagged}(x) = y\ \mathsf{then}\ z.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$**:** In this case $H_2 = H_1$ and $H(x) = \langle x', v, \vec{v}\rangle$. Since $\vdash_{\mathcal{S}} P_1 : \tau$ there must be a Γ such that $\vdash_{\mathcal{S}} H_1 : \Gamma$ and $\Gamma \vdash_{\mathcal{S}} E[I] : \tau$. By a fairly straight forward induction of the structure of E, it can be shown that there exists τ' such that $\Gamma \vdash_{\mathcal{S}} I : \tau'$. By inspection of tagging language typing rules, it must be that the tag if rule was used and hence that $\Gamma \vdash_{\mathcal{S}} \mathsf{tagged}(x) : \mathsf{tagged}$ and that $\Gamma \vdash_{\mathcal{S}} e_2 : \mathsf{tag}(\sigma)$ for some σ. By further inspection of the rules, it must be that the tagged rule was used and hence that $\Gamma \vdash_{\mathcal{S}} x : \langle\mathsf{tag}(\tau_1), \tau_1\rangle$. By inspection of the rules it must be that $\Gamma(x') = \mathsf{tag}(\tau_1)$ and hence $\Gamma \vdash_{\mathcal{S}} x' : \mathsf{tag}(\tau_1)$. Thus all of the hypotheses to lemma C.4 are satisfied. Note that $[\![I]\!]_{\text{exp}}$ is:

$$\begin{array}{l} \mathsf{let}\ x_1 = \mathsf{pack}[[\![\tau_1]\!]_{\text{type}}, x]\ \mathsf{as}\ [\![\mathsf{tagged}]\!]_{\text{type}}\ \mathsf{and}\ x_2 = y\ \mathsf{in}\ \mathsf{unpack}[\alpha, y_1] = x_1\ \mathsf{in} \\ \mathsf{if?}\ \mathit{tagchk}(y_1, x_2, \alpha, [\![\sigma]\!]_{\text{type}})\ y_1.1\ \mathsf{then}\ z.[\![b_1]\!]_{\text{exp}}\ \mathsf{else}\ [\![b_2]\!]_{\text{exp}}\ \mathsf{fi} \end{array}$$

So:

$$
\begin{array}{rl}
& \mathsf{let}\ [\![H_1]\!]_{\mathrm{heap}}\ \mathsf{in}\ [\![E]\!]_{\mathrm{ectxt}}[[\![I]\!]_{\mathrm{exp}}] \\
\mapsto & \mathsf{let}\ [\![H_1]\!]_{\mathrm{heap}}\ \mathsf{in} \\
& [\![E]\!]_{\mathrm{ectxt}}[\mathsf{unpack}[\alpha, y_1] = \mathsf{pack}[[\![\tau_1]\!]_{\mathrm{type}}, x]\ \mathsf{as}\ [\![\mathsf{tagged}]\!]_{\mathrm{type}}\ \mathsf{in} \\
& \qquad\qquad \mathsf{if?}\ \mathit{tagchk}(y_1, y, \alpha, [\![\sigma]\!]_{\mathrm{type}})\ y_1.1\ \mathsf{then}\ z.[\![b_1]\!]_{\mathrm{exp}}\ \mathsf{else}\ [\![b_2]\!]_{\mathrm{exp}}\ \mathsf{fi}] \\
\mapsto & \mathsf{let}\ [\![H_1]\!]_{\mathrm{heap}}\ \mathsf{in} \\
& [\![E]\!]_{\mathrm{ectxt}}[\mathsf{if?}\ \mathit{tagchk}(x, y, [\![\tau_1]\!]_{\mathrm{type}}, [\![\sigma]\!]_{\mathrm{type}})\ x.1\ \mathsf{then}\ z.[\![b_1]\!]_{\mathrm{exp}}\ \mathsf{else}\ [\![b_2]\!]_{\mathrm{exp}}\ \mathsf{fi}] \\
\mapsto & \mathsf{let}\ [\![H_1]\!]_{\mathrm{heap}}\ \mathsf{in} \\
& [\![E]\!]_{\mathrm{ectxt}}[\mathsf{if?}\ \mathit{tagchk}(x, y, [\![\tau_1]\!]_{\mathrm{type}}, [\![\sigma]\!]_{\mathrm{type}})\ x'\ \mathsf{then}\ z.[\![b_1]\!]_{\mathrm{exp}}\ \mathsf{else}\ [\![b_2]\!]_{\mathrm{exp}}\ \mathsf{fi}] \\
\mapsto & \mathsf{let}\ [\![H_1]\!]_{\mathrm{heap}}\ \mathsf{in}\ [\![E]\!]_{\mathrm{ectxt}}[\mathsf{if?}\ f\ \mathsf{then}\ z.[\![b_1]\!]_{\mathrm{exp}}\ \mathsf{else}\ [\![b_2]\!]_{\mathrm{exp}}\ \mathsf{fi}]
\end{array}
$$

Where $f = \mathsf{some}(v)$ if $\mathit{tagchk}^H(x_1, y)$ and $f = \mathsf{none}^{[\![\sigma]\!]_{\mathrm{type}}}$ otherwise. There are two cases: $\mathit{tagchk}^H(x_1, y)$ or not. Assume the former. Then $f = \mathsf{some}(v)$ and $e = b_2[z := v]$ and:

$$
\begin{array}{rl}
& \mathsf{let}\ [\![H_1]\!]_{\mathrm{heap}}\ \mathsf{in}\ [\![E]\!]_{\mathrm{ectxt}}[\mathsf{if?}\ f\ \mathsf{then}\ z.[\![b_1]\!]_{\mathrm{exp}}\ \mathsf{else}\ [\![b_2]\!]_{\mathrm{exp}}\ \mathsf{fi}] \\
\mapsto & \mathsf{let}\ [\![H_1]\!]_{\mathrm{heap}}\ \mathsf{in}\ [\![E]\!]_{\mathrm{ectxt}}[[\![b_1]\!]_{\mathrm{exp}}[z := v]] \\
= & \mathsf{let}\ [\![H_2]\!]_{\mathrm{heap}}\ \mathsf{in}\ [\![E]\!]_{\mathrm{ectxt}}[[\![b_1[z := v]]\!]_{\mathrm{exp}}] \\
= & \mathsf{let}\ [\![H_2]\!]_{\mathrm{heap}}\ \mathsf{in}\ [\![E]\!]_{\mathrm{ectxt}}[[\![e]\!]_{\mathrm{exp}}]
\end{array}
$$

Note that lemma C.1 was used in the second to last step. Now assume the other case. Then $f = \mathsf{none}^{[\![\sigma]\!]_{\mathrm{type}}}$ and $e = [\![b_2]\!]_{\mathrm{exp}}$ and:

$$
\begin{array}{rl}
& \mathsf{let}\ [\![H_1]\!]_{\mathrm{heap}}\ \mathsf{in}\ [\![E]\!]_{\mathrm{ectxt}}[\mathsf{if?}\ f\ \mathsf{then}\ z.[\![b_1]\!]_{\mathrm{exp}}\ \mathsf{else}\ [\![b_2]\!]_{\mathrm{exp}}\ \mathsf{fi}] \\
\mapsto & \mathsf{let}\ [\![H_1]\!]_{\mathrm{heap}}\ \mathsf{in}\ [\![E]\!]_{\mathrm{ectxt}}[[\![b_2]\!]_{\mathrm{exp}}] \\
= & \mathsf{let}\ [\![H_2]\!]_{\mathrm{heap}}\ \mathsf{in}\ [\![E]\!]_{\mathrm{ectxt}}[[\![e]\!]_{\mathrm{exp}}]
\end{array}
$$

□

Theorem C.6 *If $\vdash_{\mathcal{S}} P : \tau$ then $P \not\mapsto$ if and only if $[\![P]\!]_{\mathrm{prog}} \not\mapsto$.*

Proof: First, by an inspection of the translation rules, a value is translated into a value, and a nonvalue is translated into a nonvalue. Second, by the soundness of the tagging language, if $P \not\mapsto$ then P has the form $\mathsf{let}\ H\ \mathsf{in}\ v$, which means $[\![P]\!]_{\mathrm{prog}}$ is a terminal configuration, which is irreducible in the target language. Third, by the type preservation lemma $\vdash_{\mathcal{T}} [\![P]\!]_{\mathrm{prog}} : [\![\tau]\!]_{\mathrm{type}}$. So if $[\![P]\!]_{\mathrm{prog}} \not\mapsto$ then by the target language type soundness $[\![P]\!]_{\mathrm{prog}}$ is a terminal configuration. Therefore P must have the form $\mathsf{let}\ H\ \mathsf{in}\ v$, which is irreducible in the tagging language. □

Putting the last two theorems together, a typeable tagging language program diverges if and only if its translation diverges, and converges to a terminal configuration if and only if its translation converges to the translation of that terminal configuration. I take this as an adequate (but overly strict) meaning for operational correctness.

D Array Optimisation

Common implementations of downcasting for single inheritance hierarchies use arrays instead of linked lists, and trade space for time. Let person by a top level tag, student a subtag of person, and graduate a subtag of student. Under the array scheme, person is implemented as an array of length one containing itself. Student is implemented as an array of length two whose second element is itself and first element is person. Graduate is implemented as an array of length three whose third element is itself, second element is student, and first element is person. In general a tag at depth i in the tag hierarchy is implemented as an array of length i whose i-th element is itself, and whose first through $i-1$-th elements are the tag's ancestors at depth 1 through $i-1$. Note that if t_1 is under t_2 in the tag hierarchy then t_2 appears in the t_1 array, and at index equal to t_2's depth. Note also that a tags depth is just the length of the array representing it. Therefore the predicate $tagchk^H(x,y)$ can be performed by comparing entry $y.\mathsf{size}$ of array x to y. The rest of this section will present a variant of the target calculus, and a formal translation corresponding to this array scheme. The array scheme requires space proportional to the number of tags times the depth of the tag hierarchy but constant time for a tag check; the scheme used in the main body of this report requires space porportional to the number of tags but time proportional to the depth of the tag hierarchy.

First, to accomodate the cyclic datastructures inherent in the array scheme, the target language heaps need to be cyclic. This is achieved by changing the typing rule for heaps to:

$$\frac{\epsilon \vdash_{\mathcal{T}} \tau_i \quad \Gamma \vdash_{\mathcal{T}} h_i : \tau_i}{\vdash_{\mathcal{T}} x_1 = h_1, \ldots, x_n = h_n : \Gamma}\ (\Gamma = x_1{:}\tau_1, \ldots, x_n{:}\tau_n)$$

Second, arrays need to be added to the language. The extended syntax is:

$$\begin{array}{lcl}
\tau, \sigma & ::= & \cdots \mid \mathsf{array}(\tau) \mid \mathsf{int} \\
e, b & ::= & \cdots \mid \mathsf{fix}\, x = \mathsf{tag}(\{e_1, \ldots, e_n\}, \tau) \mid \mathsf{fix}\, x = \mathsf{tag}(\mathsf{extend}(e_1, e_2), \tau) \mid \\
 & & e.\mathsf{size} \mid e_1[e_2] \Rightarrow x.b_1 \text{ else } b_2 \\
v & ::= & \cdots \mid i \\
E & ::= & \cdots \mid \mathsf{fix}\, x = \mathsf{tag}(\{\vec{v}, E, \vec{e}\}, \tau) \mid \mathsf{fix}\, x = \mathsf{tag}(\mathsf{extend}(E, e), \tau) \mid \\
 & & \mathsf{fix}\, x = \mathsf{tag}(\mathsf{extend}(v, E), \tau) \mid E.\mathsf{size} \mid E[e] \Rightarrow x.b_1 \text{ else } b_2 \mid v[E] \Rightarrow x.b_1 \text{ else } b_2 \\
h & ::= & \cdots \mid \mathsf{tag}(\{v_1, \ldots, v_n\}, \tau)
\end{array}$$

Arrays are created by one of two operations. As I will use arrays only for implementing tags, both operations are special recursive definition forms that introduce tags. The expression $\mathsf{fix}\, x = \mathsf{tag}(a, \tau)$ creates a new array a which will be used as a tag for type τ; x may be used in a to refer to the new array. The first form, $a = \{e_1, \ldots, e_n\}$, creates an array of size n with elements e_1 through e_n. The second form, $a = \mathsf{extend}(e_1, e_2)$, creates an array that is a copy of array e_1 but with an additional element whose value is e_2. An array's size is obtained with $e.\mathsf{size}$. The subscript operation is a little unusual, in that, it combines an explicit bounds with the subscript operation. The expression $e_1[e_2] \Rightarrow x.b_1$ else b_2 evalutes e_1 to an array and e_2 to an integer, then if e_2 is within the array bounds, x is bound to the desired element and b_1 is evaluated, otherwise b_2 is evaluated.

The additional reduction rules necessary to formalise the operations are:

I	e	H'	Side Conditions
$\mathsf{fix}\, x = \mathsf{tag}(\{v_1, \ldots, v_n\}, \tau)$	x	$H, x = h$	$x \notin H, h = \mathsf{tag}(\{v_1, \ldots, v_n\}, \tau)$
$\mathsf{fix}\, x = \mathsf{tag}(\mathsf{extend}(y, v), \tau)$	x	$H, x = h$	$x \notin H, h = \mathsf{tag}(\{v_1, \ldots, v_n, v\}, \tau)$ $H(y) = \mathsf{tag}(\{v_1, \ldots, v_n\}, \sigma)$
$x.\mathsf{size}$	n	H	$H(x) = \mathsf{tag}(\{v_1, \ldots, v_n\}, \sigma)$
$x[i] \Rightarrow x.b_1\ \mathsf{else}\ b_2$	$b_1[x := v_i]$	H	$H(x) = \mathsf{tag}(\{v_1, \ldots, v_n\}, \sigma)$ $1 \leq i \leq n$
$x[i] \Rightarrow x.b_1\ \mathsf{else}\ b_2$	b_2	H	$H(x) = \mathsf{tag}(\{v_1, \ldots, v_n\}, \sigma)$ $\neg(1 \leq i \leq n)$

The additional typing rules are:

$$\frac{\Delta \vdash_{\mathcal{T}} \tau_1 \leq \tau_2}{\Delta \vdash_{\mathcal{T}} \mathsf{array}(\tau_1) \leq \mathsf{array}(\tau_2)}$$

$$\frac{\Delta \vdash_{\mathcal{T}} \sigma \quad \Delta; \Gamma, x{:}\sigma \vdash_{\mathcal{T}} e_i : \tau'}{\Delta; \Gamma \vdash_{\mathcal{T}} \mathsf{fix}\, x = \mathsf{tag}(\{e_1, \ldots, e_n\}, \tau) : \sigma}\ (x \notin \Gamma; \sigma = \mathsf{tag}^{\circ}(\tau, \mathsf{array}(\tau')))$$

$$\frac{\Delta \vdash_{\mathcal{T}} \sigma \quad \Delta; \Gamma, x{:}\sigma \vdash_{\mathcal{T}} e_1 : \mathsf{array}(\tau') \quad \Delta; \Gamma, x{:}\sigma \vdash_{\mathcal{T}} e_2 : \tau'}{\Delta; \Gamma \vdash_{\mathcal{T}} \mathsf{fix}\, x = \mathsf{tag}(\mathsf{extend}(e_1, e_2), \tau) : \sigma}\ (x \notin \Gamma; \sigma = \mathsf{tag}^{\circ}(\tau, \mathsf{array}(\tau')))$$

$$\frac{\Delta; \Gamma \vdash_{\mathcal{T}} e : \mathsf{array}(\tau)}{\Delta; \Gamma \vdash_{\mathcal{T}} e.\mathsf{size} : \mathsf{int}} \qquad \frac{}{\Delta; \Gamma \vdash_{\mathcal{T}} i : \mathsf{int}}$$

$$\frac{\Delta; \Gamma \vdash_{\mathcal{T}} e_1 : \mathsf{array}(\sigma) \quad \Delta; \Gamma \vdash_{\mathcal{T}} e_2 : \mathsf{int} \quad \Delta; \Gamma, x{:}\sigma \vdash_{\mathcal{T}} b_1 : \tau \quad \Delta; \Gamma \vdash_{\mathcal{T}} b_2 : \tau}{\Delta; \Gamma \vdash_{\mathcal{T}} e_1[e_2] \Rightarrow x.b_1\ \mathsf{else}\ b_2 : \tau}$$

$$\frac{\epsilon \vdash_{\mathcal{T}} \tau \quad \epsilon; \Gamma \vdash_{\mathcal{T}} v_i : \sigma}{\Gamma \vdash_{\mathcal{T}} \mathsf{tag}(\{v_1, \ldots, v_n\}, \tau) : \mathsf{tag}^{\circ}(\tau, \mathsf{array}(\sigma))}$$

This extended language is type sound. The proof is very similar to the one given in Appendix A but has additional cases for the new constructs. These constructs, while presented in an unusual way, are standard and have been proven sound before.

Third, I need to give a new translation that uses the array scheme. In fact, most of the old translation can be reused, I just need to change some key items. At the type level only the auxilary functions need to change, and they now reflect the array data structure rather than the linked list data structure. At the term level the translation of newtag, subtag, and the auxilary tag check function need to change to reflect the new datastructure.

$$\begin{array}{lcl}
\mathit{tag}(\phi, \tau) & = & \mathsf{tag}^{\phi}(\tau, \mathsf{array}(\mathit{tag}'(\tau))) \\
\mathit{tag}'(\tau) & = & \mathsf{rec}\ \alpha.\mathsf{tag}^{-}(\tau, \mathsf{array}(\alpha)) \\
[\![\mathsf{newtag}(\tau)]\!]_{\text{exp}} & = & \mathsf{fix}\, x = \mathsf{tag}(\{\mathsf{roll}^{\mathit{tag}'([\![\tau]\!]_{\text{type}})}(x)\}, [\![\tau]\!]_{\text{type}}) \\
[\![\mathsf{subtag}(e, \tau)]\!]_{\text{exp}} & = & \mathsf{fix}\, x = \mathsf{tag}(\mathsf{extend}([\![e]\!]_{\text{exp}}, \mathsf{roll}^{\mathit{tag}'([\![\tau]\!]_{\text{type}})}(x)), [\![\tau]\!]_{\text{type}}) \\
\mathit{tagchk}(y_1, y_2, \tau, \sigma) & = & \mathsf{fix}\, y_3(y_4 : \mathit{tag}(-, \tau)) : \sigma?. \\
& & \quad y_4[y_2.\mathsf{size}] \Rightarrow \\
& & \qquad y_5.\mathsf{if}\ \mathsf{unroll}(y_5) = y_2\ \mathsf{then}\ \mathsf{some}(y_1.2)\ \mathsf{else}\ \mathsf{none}^{\sigma}\ \mathsf{fi} \\
& & \quad \mathsf{else}\ \mathsf{none}^{\sigma}
\end{array}$$

The translation of heaps is trickier, because in translating a subtag, (τ, y), all the ancestors of y are needed. To provide this information, the heap being constructed is threaded through the translation of heap values.

$$
\begin{array}{lcl}
[\![x = (\tau, \epsilon)]\!]^H_{\text{hval}} & = & x = \mathsf{tag}(\{\mathsf{roll}^{tag'([\![\tau]\!]_{\text{type}})}(x)\}, [\![\tau]\!]_{\text{type}}) \\
[\![x = (\tau, y)]\!]^H_{\text{hval}} & = & x = \mathsf{tag}(\{v_1, \ldots, v_n, \mathsf{roll}^{tag'([\![\tau]\!]_{\text{type}})}(x)\}, [\![\tau]\!]_{\text{type}}) \\
 & & \text{where } H(y) = \mathsf{tag}(\{v_1, \ldots, v_n\}, \sigma) \\
[\![x = \langle v_1, \ldots, v_n \rangle]\!]^H_{\text{hval}} & = & x = \langle [\![v_1]\!]_{\text{exp}}, \ldots, [\![v_n]\!]_{\text{exp}} \rangle \\
[\![x_1 = h_1, \ldots, x_n = h_n]\!]_{\text{heap}} & = & H_n \\
 & & \text{where } H_0 = \epsilon, H_{i+1} = H_i, [\![h_{i+1}]\!]^{H_i}_{\text{hval}}
\end{array}
$$

The new translation is type preserving and operationally correct. The proofs are very similar to those given in Appendices B and C. In the case of type preservation, the proof of lemma B.1, the proof of lemma B.4, the cases for newtag and subtag in the proof of theorem B.6, the statement and proof of theorem B.7, and the proof of theorem B.8 change in a straighforward manner to reflect the array datastructure.

The proof of operational correctness involves more substantial changes. Lemma C.3 is replaced by the two lemmas below. The proof of lemma C.4, and the cases for newtag and subtag in the proof of theorem C.5 change to reflect the new datastructure.

Lemma D.1 *If* $\vdash_{\mathcal{S}} H : \Gamma$ *and* $\Gamma \vdash_{\mathcal{S}} x : \mathsf{tag}(\tau_x)$ *then there exists an* $n \geq 1$ *such that there exists* $w_1, \ldots, w_n$*, and* $\tau_1, \ldots, \tau_n$*, such that* $x = w_n$*,* $\forall 1 < i \leq n : H(w_i) = (\tau_i, w_{i-1})$*, and* $H(w_1) = (\tau_1, \epsilon)$*. Define* $tchain^H(x) = \langle w_1, \ldots, w_n \rangle$*.*

Lemma D.2 *If* $\vdash_{\mathcal{S}} H : \Gamma$*,* $\Gamma \vdash_{\mathcal{S}} x : \mathsf{tag}(\tau_x)$*, and* $\Gamma \vdash_{\mathcal{S}} y : \mathsf{tag}(\tau_y)$ *then* $tagchk^H(x, y)$ *if and only if* $w_{|tchain^H(y)|} = y$ *where* $tchain^H(x) = \langle w_1, \ldots, w_n \rangle$*.*

Both the array scheme and the linked list scheme are examples of a more general scheme. They both use unique pointers to represent tags and some sort of datastructure to capture the hierarchy. Tag checking is performed by searching the datastructure for one or more pairs of tags to compare, and the tag check succeeds only if one of these pointer comparisons succeed. Any such scheme should be implementable in a variant of the target language suitable extended with constructs to implement the datastructure. Such schemes include a number of efficient implementations of multiple inheritance tag hierarchies.