A Generic Usage Analysis with Subeffect Qualifiers

Jurriaan Hage Stefan Holdermans Arie Middelkoop

Department of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
{jur,stefan,ariem}@cs.uu.nl

Abstract

Sharing analysis and uniqueness typing are static analyses that aim at determining which of a program's objects are to be used at most once. There are many commonalities between these two forms of usage analysis. We make their connection precise by developing an expressive generic analysis that can be instantiated to both sharing analysis and uniqueness typing. The resulting system, which combines parametric polymorphism with effect subsumption, is specified within the general framework of qualified types, so that readily available tools and techniques can be used for the development of implementations and metatheory.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Theory

Keywords sharing analysis, uniqueness typing, type and effect systems, qualified types

1. Introduction

Sharing analysis and *uniqueness typing* are static analyses that both aim at determining which objects in a functional program are guaranteed to be used at most once and, dually, which objects may be used more than once.

It has been recognized that there are many overlaps between the techniques used in the specification of type-based sharing analyses and the definition of calculi with uniqueness typing (Wansbrough and Peyton Jones 1999). In this paper we make the connection between these analyses precise by presenting a single, generic type and effect system that may be instantiated to both sharing analysis and uniqueness typing. Our specific contributions are the following:

- We show how the differences between sharing analysis and uniqueness typing amount to inversion of a subsumption relation between usage properties. In our approach, the specific direction of this relation is the sole parameter of a generic system that captures the commonalities between the two analyses (Section 3.1).
- We present an explicitly typed calculus that serves as a target for both sharing analysis and uniqueness typing, and equip it with a type and effect system that features type polymorphism as well as effect polymorphism (Section 4). A generic analysis is then developed by conservatively extending the well-known Hindley-Milner typing discipline (Milner 1978) and defining a type-preserving translation from implicitly typed source terms into explicitly typed target terms (Section 5). Our analysis comes with a correctness result (Section 4.4) and a principal-type property that facilitates the derivation of an incremental inference algorithm (Section 5.3).
- A notable characteristic of our type and effect system is that it provides parametric polymorphism and *subeffecting* as its only means of subsumption. In particular, we do not extend the subsumption relation between usage properties to a subsumption relation between types, as is often done in related systems (Barendsen and Smetsers 1993; Wansbrough and Peyton Jones 1999; Gustavsson 1999). We argue that this specific design choice has at most a modest impact on the expressiveness of our analyses, while it allows for an implementation that is considerably less complicated than it would have been in the presence of a full subtype relation (Section 3.5).
- Subeffecting is incorporated into our system as a form of ad-hoc polymorphism. This enables us to model all use of subsumption within the framework of *qualified types* (Jones 1994), so that implementations and metatheory can, to a large extent, be developed by reusing tools and techniques that are readily available for this framework (Section 3.3).

At this point, our use of terminology deserves some clarification. Some authors (see Section 6 for a discussion of related work) use the term "usage analysis" for what we call "sharing analysis". We, in turn, reserve the name "usage analysis" for a general category of analyses that contains both sharing analysis and uniqueness typing. Our choice for the term "sharing analysis" is motivated by our inclination to stress the complementary nature of sharing and uniqueness. This complementariness is elaborated upon in the following section.

2. Two Notions of Usage

In this section, we briefly introduce sharing analysis (Section 2.1) and uniqueness typing (Section 2.2), and highlight the differences between these two analyses (Section 2.3). For a more thorough introduction to type-based sharing analysis, the reader is referred to Turner et al. (1995). An introduction to uniqueness typing is given by Barendsen and Smetsers (1993). For further references, see Section 6.

Both sharing analysis and uniqueness typing are primarily targeted at languages with lazy, i.e., call-by-need, evaluation strategies, such as Haskell (Peyton Jones 2003) and Clean (Plasmeijer

ICFP'07, October 1–3, 2007, Freiburg, Germany.

and Van Eekelen 1998). Variations, however, may also be of use in other contexts (e.g., Turner et al. 1995; Mogensen 1998; Kobayashi 1999). In this paper, though, we limit ourselves to call-by-need languages.

2.1 Sharing Analysis

Call-by-need evaluation is usually implemented by means of updatable closures. For instance, in the program

$$\textbf{let } x = 2 + 3 \textbf{ in } x + x,$$

the variable x represents a closure that initially contains the subterm $2 + 3$. After the first demand for x, this closure is updated with the value 5. The second demand for x now immediately retrieves the computed value and so the evaluation of $2 + 3$ is effectively shared between the occurences of x in the body of the local definition.

However, in the following program,

$$\textbf{let } y = 2 + 3 \textbf{ in } 2 * y,$$

the value of the variable y is demanded only once and, hence, its evaluation will not be shared. Therefore, updating the closure for y makes little sense.

In general, it is unnecessary to update a closure if its value is demanded no more than once. Instead, after it has produced its value, such a closure can be removed from memory altogether. Now, the goal of a sharing analysis is to determine

> for each of a program's subterms, whether or not its evaluation may be shared.

The information obtained from sharing analysis can, other than for avoiding unnecessary updates, also be used to faciliate a series of program transformations (Turner et al. 1995; Wansbrough and Peyton Jones 1999; Gustavsson and Sands 1999).

A sharing analysis typically classifies terms into two groups: those that are guaranteed to be used at most once and those that may be used more than once. Type-based sharing analyses record these classifications in type derivations. For the first example above, for instance, such an analysis produces the typing

$$x :^{\omega} \mathsf{Int}.$$

Here, ω indicates that the evaluation of x may be shared. For the second example we have

$$y :^{1} \mathsf{Int},$$

indicating that y is used at most once. In type-based program analysis, annotations such as 1 and ω are often referred to as *effects* (Gifford 1986; Talpin and Jouvelot 1994; Nielson and Nielson 1999). In the specific context of usage analysis, we call them usage effects or usage annotations.

Usage effects also appear within function types. Consider, for instance, the function

$$\mathit{double}\ x = 2 * x$$

and its typing

$$\mathit{double} :^{\omega} \mathsf{Int}^{1} \to \mathsf{Int}^{\omega}.$$

The domain and codomain of the function type are annotated with usage effects. The effect 1 on the domain indicates that *double* uses its argument at most once; the effect ω on the codomain indicates that results produced by *double* may be used more than once. The remaining ω ranges over the whole typing and expresses that the function itself may be shared.

2.2 Uniqueness Typing

Uniqueness typing is concerned with maintaining referential transparency in the presence of destructive updates or, more general, side-effecting computations. Its purpose is to determine

> for each of a program's subterms, whether or not its value is required to be *unique*.

Here, uniqueness means that a value is used at most once.

As an example, consider a function *fPutChar* that appends a character to a file and returns the updated file:

$$\mathit{fPutChar} : \mathsf{Char} \to \mathsf{File} \to \mathsf{File}.$$

Such a function does not violate referential transparency if it is granted private access to its second argument. Systems with uniqueness typing allow for this demand to be expressed in the type of *fPutChar*:

$$\mathit{fPutChar} :^{\omega} \mathsf{Char}^{\omega} \to (\mathsf{File}^{1} \to \mathsf{File}^{1})^{\omega}.$$

The 1-annotations in this typing indicate that *fPutChar* needs to be passed a unique, i.e., nonshared, file argument and that the file it produces is then also unique. The ω-annotations indicate that uniqueness of the character argument is not required, and that the function itself and its partial applications may be used more than once.

Often, uniqueness typing enforces objects that are subjected to side-effecting, such as files, to be passed around *single-threadedly*. For instance, the following program is ill-typed,

$$\begin{array}{l}\textbf{let } f = \mathit{readFile}\ \texttt{"DATA"}\\ \textbf{in } (\mathit{fPutChar}\ \texttt{'O'}\ f, \mathit{fPutChar}\ \texttt{'K'}\ f),\end{array}$$

because the file f is used twice in the body of the local definition. In contrast, the program

$$\begin{array}{l}\textbf{let } f = \mathit{readFile}\ \texttt{"DATA"}\\ \textbf{in } \mathit{fPutChar}\ \texttt{'K'}\ (\mathit{fPutChar}\ \texttt{'O'}\ f)\end{array}$$

is type-correct: the file f is used only once.

2.3 Divergence

Before we look into the differences between sharing analysis and uniqueness typing, we point out that there is a great deal of commonality between the two: both analyses keep track of how many times values are used, and both do so by classifying terms into unique (single-use) terms and shared (multi-use) terms. Furthermore, both analyses may record usage properties in the types of terms.

To see where divergence arises, consider again the function *double*, its sharing analysis

$$\mathit{double} :^{\omega} \mathsf{Int}^{1} \to \mathsf{Int}^{\omega},$$

and its use in the program

$$\textbf{let } x = 2 + 3 \textbf{ in } (\mathit{double}\ x) * x.$$

Here, x is used twice in the body of the local definition and, hence, its evaluation is shared. Still, as far as sharing analysis is concerned, it is all right to pass x to *double*, even though the parameter type of the latter is annotated with 1. Indeed, if we pass *double* an updatable closure, nothing bad happens. During the evaluation of *double x* the closure of x is updated with the result of the addition (after all, we passed in an *updatable* closure) and when the closure is entered for the second time (i.e., when the right operand of the multiplication is demanded), it immediately delivers 5. In summary: in sharing analysis, a function with a unique parameter type may just as well be passed a shared closure as argument.

However, passing a unique argument to a function with a shared parameter type is not allowed. To see why, imagine what would happen if we pass a nonupdatable closure to a function that uses its argument more than once. The first use of the argument then leaves the closure unupdated and, as a result, consecutive accesses have to re-evaluate the contained term. Worse, an optimising compiler

may well emit code that, after what is assumed to be its only use, reclaims the memory occupied by the closure and so introduce unexpected run-time errors.

In uniqueness typing, things are just the other way around. In a function's uniqueness type, a 1-annotation in parameter position specifies that the function requires private access to its argument; passing in a shared value may then compromise referential transparency. On the other hand, passing a unique value to a function with a shared parameter type is fine: an ω-parameter does not mean that a function denies private access, it rather indicates that uniqueness is not required.

To recapitulate: in sharing analysis, a shared parameter type expresses a demand (i.e., to pass in an updatable closure) on the caller of a function, while a unique parameter type expresses the absence of such a demand; in uniqueness typing, a demand (i.e., to pass in a unique argument value) is expressed by a unique parameter type and absence of this demand is expressed by a shared parameter type. In the next section, we demonstrate how this distinction between sharing analysis and uniqueness typing is addressed by our generic usage analysis.

3. A Generic Approach

In this section, we highlight the most distinguishing features of our generic type and effect system: its use of subeffecting (Section 3.1) and polyvariance (Section 3.2), its use of qualified types (Sections 3.3 and 3.4), and the absence of a subtype relation (Section 3.5).

3.1 Subeffecting

In Section 2.3 we saw that, in sharing analysis, shared closures may be passed safely to functions that use their argument at most once as well as to functions that may use their arguments more than once. In our system, this is expressed in terms of a *subeffect* relation. To enable subeffecting, we impose an ordering on effects: $1 \sqsubset \omega$.

For sharing analysis, subeffecting now expresses that it is safe to replace the effect component of an analysis by a smaller (or equal) effect value:

$$\frac{\vdash t :^{\varphi'} \tau \quad \vdash \varphi \sqsubseteq \varphi'}{\vdash t :^{\varphi} \tau} \qquad \text{(T-SubDown)}$$

This subeffecting rule is used to "prepare" the effects of function arguments. For instance, if we have established that the value associated with an integer variable x may be used more than once,

$$x :^{\omega} \mathsf{Int},$$

and x is passed to a function that uses its argument at most once, say *double*,

$$\mathit{double} :^{\omega} \mathsf{Int}^1 \to \mathsf{Int}^{\omega},$$

then T-SubDown may be used to make the effect of x compatible with the parameter effect of *double*:

$$x :^{1} \mathsf{Int}.$$

Of course, rule T-SubDown is not safe for uniqueness typing. In uniqueness typing, shared values cannot be passed to functions that expect unique arguments, but, in contrast, it is no problem to pass unique values to functions with ω-parameters. So, for uniqueness typing, we need a subeffecting rule that allows effect components to be enlarged:

$$\frac{\vdash t :^{\varphi'} \tau \quad \vdash \varphi \sqsupseteq \varphi'}{\vdash t :^{\varphi} \tau} \qquad \text{(T-SubUp)}$$

Note how rules T-SubDown and T-SubUp differ in the direction of the effect inequality. Interestingly, in our formulation, this difference in the subeffecting rules is the only essential distinction between sharing analysis and uniqueness typing. In our generic usage analysis, this distinction is made explicit in a single rule for subeffecting that is parameterized by the direction of the effect inequality, much like:

$$\frac{\vdash t :^{\varphi'} \tau \quad \vdash \varphi \diamond \varphi'}{\vdash t :^{\varphi} \tau} \qquad \text{(T-SubGen)}$$

Here, the symbol $\diamond$ denotes the parameter of a type and effect system that subsumes sharing analysis (if $\diamond$ is instantiated with $\sqsubseteq$) as well as uniqueness typing (if $\diamond$ is instantiated with $\sqsupseteq$). Moreover, $\diamond$ is the only parameter of the generic analysis and the rule for subeffecting is the only rule in the system that explicitly invokes the parameter.

It is important to realize that the difference between sharing analysis and uniqueness typing does not amount to a reversal of the ordering on effects: our system is parameterized by a specific use of the ordering relation in the rule for subeffecting, not by the ordering itself. We show why this is important below in Section 3.4, when we discuss the containment restriction.

3.2 Polyvariance

As shown in the previous subsection, subeffecting is used to adapt the effect of a function argument, so that it matches the parameter effect of the function it is passed to. Dually, our generic analysis also provides a means to adapt the type of a function to fit the argument it is applied to, namely, *effect polymorphism* or *polyvariance*.

For instance, the fact that, in sharing analysis, a function that uses its argument at most once does not put any demand on the usage property of its argument, can be expressed by assigning such a function a type that is polymorphic in the effect of its parameter, e.g,

$$\mathit{double} :^{\omega} \forall \mathsf{p}.\ \mathsf{Int}^{\mathsf{p}} \to \mathsf{Int}^{\omega}.$$

Here, p is an *effect variable* that may be instantiated with both 1 and ω, which is consistent with our observation that a function with a single-use parameter, such as *double*, can be passed a single-use argument as well as a multi-use argument.

A function can be polyvariant not only in its parameter, but also in its result. For example, whether the result of *double* is used at most once or more than once depends on the context in which it is produced. Hence, we abstract over the result effect and yield

$$\mathit{double} :^{\omega} \forall \mathsf{p}\ \mathsf{q}.\ \mathsf{Int}^{\mathsf{p}} \to \mathsf{Int}^{\mathsf{q}}.$$

The type of *double* can now be instantiated to $\mathsf{Int}^1 \to \mathsf{Int}^1$, $\mathsf{Int}^1 \to \mathsf{Int}^{\omega}$, $\mathsf{Int}^{\omega} \to \mathsf{Int}^1$, and $\mathsf{Int}^{\omega} \to \mathsf{Int}^{\omega}$, which are all indeed valid analysis results for *double*.

3.3 Subeffect Qualification

Our use of subeffecting and polyvariance allows our analyses to be, to a large extent, *context sensitive*: they allow for the type and effect of a term to be adapted to the context in which the term is used. To see why it is beneficial to include both subeffecting and polyvariance in a single system, let us look into the analysis of higher-order functions.

As an example of a higher-order function, consider the function *apply*,

$$\mathit{apply}\ f\ x = f\ x,$$

that applies its first argument to its second. Let us assume for now that *apply* only operates on shared integer functions. Now, how should we annotate the type of the second parameter x? A moment's reflection reveals that which effects are valid for x depends on the parameter effect of f. In sharing analysis, if f has a 1-parameter,

then both 1 and ω are valid annotations for x; if f has an ω-parameter, then the type of x should actually be annotated with ω. Similarly, in uniqueness typing, an ω-annotation for the parameter of f means that x can be annotated with 1 as well as ω, while if f requires a unique argument, then x should in fact be unique. These observations are compactly captured by a so-called *qualified type*:

$$\mathit{apply} :^{\omega} \forall \mathsf{p}_1\ \mathsf{p}_2\ \mathsf{q}\ \mathsf{r}.\ (\mathsf{p}_1 \diamond \mathsf{p}_2) \Rightarrow (\mathsf{Int}^{\mathsf{p}_1} \to \mathsf{Int}^{\mathsf{q}})^{\omega} \to (\mathsf{Int}^{\mathsf{p}_2} \to \mathsf{Int}^{\mathsf{q}})^{\mathsf{r}}.$$

In this type, the predicate $\mathsf{p}_1 \diamond \mathsf{p}_2$ denotes a *subeffect qualifier*; it expresses that the only valid simultaneous instantiations for p_1 and p_2 are those that satisfy the inequality $\mathsf{p}_1 \diamond \mathsf{p}_2$.

Qualified types (Jones 1994) are a general framework for combining parametric and ad-hoc polymorphism in type systems. This framework is best known for providing a theoretical foundation for Haskell's type classes (Kaes 1988; Wadler and Blott 1989). Other applications include subtyping, extensible records, and implicit parameter passing (Lewis et al. 2000).

Using qualified types in our analysis allows us to use familiar tools and techniques for the development of implementations and metatheories. For example, if we extend an existing Haskell compiler with an implementation of our usage analysis, we are able to reuse a great deal of infrastructure that is already present for the support of type classes. Moreover, since the theory of qualified types is formulated independently from the exact form of the predicates that make up the qualifier language, it is expected that our approach can also be applied to other program analyses, such as binding-time analysis and strictness analysis.

3.4 Containment

The typing of the function *apply* in the previous subsection demonstrates the use of subeffect qualifiers in our system. However, the given analysis of *apply* is still rather imprecise, due to our assumption that *apply* only operates on shared integer functions. Indeed, a higher degree of context sensitivity can be achieved by abandoning this assumption and abstracting over the type and usage property of the function argument. Abstracting over the type is straightforward and amounts to plain type polymorphism. Abstracting over the usage property is more involved here, since we have to deal with the possibility that *apply* is partially applied.

To see why, suppose that *apply* is applied to a function argument f only: if such a partial application *apply* f is used more than once, then so is the argument f. So, abstracting over the usage property of f, we have to be careful to maintain a relationship between the effect for f and the effect for partial applications of *apply*. We do so by introducing an additional qualifier $\mathsf{r}_1 \sqsupseteq \mathsf{r}_2$ that specifies that the function argument is used as least as often as the associated partial application:

$$\mathit{apply} :^{\omega} \forall \mathsf{a}\ \mathsf{b}\ \mathsf{p}_1\ \mathsf{p}_2\ \mathsf{q}\ \mathsf{r}_1\ \mathsf{r}_2.\ (\mathsf{p}_1 \diamond \mathsf{p}_2, \mathsf{r}_1 \sqsupseteq \mathsf{r}_2) \Rightarrow (\mathsf{a}^{\mathsf{p}_1} \to \mathsf{b}^{\mathsf{q}})^{\mathsf{r}_1} \to (\mathsf{a}^{\mathsf{p}_2} \to \mathsf{b}^{\mathsf{q}})^{\mathsf{r}_2}.$$

What we see here is actually an instance of a more general scheme: if a value is contained within a structure, we must assume that it is used at least as often as the containing structure. This phenomenon is called the *containment restriction*; Barendsen and Smetsers (1993) call it *uniqueness propagation*.

Note that the second qualifier in the type of *apply* is expressed in terms of the relation $\sqsupseteq$. In particular, it is not expressed in terms of the parameter $\diamond$. The containment restriction applies to both sharing analysis and uniqueness typing and in both cases it is expressed in terms of the ordering $1 \sqsubset \omega$. So, here it is crucial that our generic system is not parameterized by the ordering on effects but, instead, by the direction of the subeffect relation.

Remark. A noteworthy subtlety arises, in uniqueness typing, from the interaction between subeffecting and the containment restriction. Assume, for instance, that containment forces the annotation for a particular partial application to be 1. Then, it may well be that subeffecting later enables us to lift this annotation to ω, effectively bypassing the containment restriction, which is, clearly, undesirable. To prevent this, Barendsen and Smetsers (1993) exclude functions from being subjected to subeffecting. For our purposes, however, this is too limiting, for it would unnecessarily affect sharing analysis as well. De Vries et al. (2007) take an interesting and more liberal approach by relaying enforcement of the containment restriction on functions to their application sites. We conjecture that this technique fits well into our system. In the remaining of this paper, however, we shall ignore the issue altogether and simply assume that subeffecting is never applied in a manner that conflicts with containment, leaving a more satisfying treatment as future work.

3.5 Subtyping

A final distinguishing aspect of our system is that it does not include subtyping. Most related systems (most notably Barendsen and Smetsers 1993; Wansbrough and Peyton Jones 1999; Gustavsson 1999) extend the ordering on effects to a shape-conformant partial ordering on types and then employ this partial ordering to derive a compliant subtype relation. However, incorporating a full subtype relation complicates the design and implementation of a usage analysis considerably, since the analysis has to deal not only with inequalities between effects, but also with inequalities between types. Moreover, the shapes of the types influence the inequalities that have to hold between effects and instantiations of type variables may therefore introduce additional effect inequalities.

In contrast, in our approach, inequalities only arise between effects and instantiating type variables does not lead to new inequalities. The resulting system, with subeffecting and polyvariance, fits nicely into the theory of qualified types and is, in practice, about as expressive as a system with subtyping, even though the inferred types are different.

For example, consider the following function,

$$\mathit{two}\ x = 2,$$

that ignores its argument and produces the constant 2. A sharing analysis based on subtyping typically assigns *two* its *least restrictive* type:

$$\mathit{two} :^{\omega} \forall \mathsf{a}.\ \mathsf{a}^{1} \to \mathsf{Int}^{\omega}.$$

Let us now assume that *two* is applied to a shared integer value x in a context in which the function application needs to be stored in a nonupdatable closure. A subtyping analysis then instantiates a to Int and invokes subtyping to coerce the type of *two* into $\mathsf{Int}^{\omega} \to \mathsf{Int}^{1}$. Our analysis, on the other hand, first assigns *two* its *most polyvariant* type,

$$\mathit{two} :^{\omega} \forall \mathsf{a}\ \mathsf{p}\ \mathsf{q}.\ \mathsf{a}^{\mathsf{p}} \to \mathsf{Int}^{\mathsf{q}},$$

and then adapts this type to the context of the call site by instantiating a with Int, p with ω, and q with 1.

4. Target Language

In this section, we present an explicitly and impredicatively typed language in the style of System F (Girard 1972; Reynolds 1974), that serves as a target language for the analysis we develop in Section 5. This target language, of which the syntax is presented in Section 4.1, can be used as a back end for both sharing analysis and uniqueness typing.

Sharing analysis and uniqueness typing are motivated by particularities in the operational behaviour of the programs they operate upon: respectively, the distinction between updatable and nonupdatable closures, and the presence of side-effecting computations.

Identifiers

$x \in \mathbf{Var}$		*(term variables)*
$\alpha \in \mathbf{TyVar}$		*(type variables)*
$\beta \in \mathbf{EffVar}$		*(effect variables)*
$\delta \in \mathbf{EvVar}$		*(evidence variables)*
$h \in \mathbf{Loc}$		*(heap locations)*

Term language

$t \in \mathbf{Tm}$	$:=$	$x \mid u$
$u \in \mathbf{PreTm}$	$:=$	$\lambda^{\varphi} x^{\varphi} : \tau.\ t \mid t\ x \mid t\ u^{\varphi}$
	$\mid$	$\lambda\alpha.\ t \mid t\ \tau \mid \lambda\beta.\ t \mid t\ \varphi \mid \lambda\delta : \pi.\ t \mid t\ \xi$
	$\mid$	$\mathbf{let}\ x^{\varphi} = t\ \mathbf{in}\ t \mid \xi\ t$
$w \in \mathbf{Whnf}$	$:=$	$\lambda^{\varphi} x^{\varphi} : \tau.\ t \mid \lambda\alpha.\ w \mid \lambda\beta.\ w \mid \lambda\delta : \pi.\ w \mid \xi\ w$
$\xi \in \mathbf{Ev}$	$:=$	$\delta \mid \imath \mid \xi \circ \xi \mid \bot \mid \top$

Type and effect language

$\tau \in \mathbf{Ty}$	$:=$	$\alpha \mid \tau^{\varphi} \to \tau^{\varphi} \mid \forall\alpha.\ \tau \mid \forall\beta.\ \tau \mid \pi \Rightarrow \tau$
$\varphi \in \mathbf{Eff}$	$:=$	$\beta \mid 1 \mid \omega$
$\pi \in \mathbf{Pred}$	$:=$	$\varphi \sqsubseteq \varphi$

Typing contexts

$\Gamma \in \mathbf{Ctx}$	$:=$	$\emptyset \mid \Gamma, x :^{\varphi} \tau \mid \Gamma, \delta : \pi$

Reduction contexts

$H \in \mathbf{Hp}$	$:=$	$\emptyset \mid H, h \mapsto (t;\ \eta)^{\varphi}$
$S \in \mathbf{Stk}$	$:=$	$\emptyset \mid S, \#h \mid S, @h \mid S, \lambda\alpha \mid S, @\tau$
	$\mid$	$S, \lambda\beta \mid S, @\varphi \mid S, \lambda\delta : \pi \mid S, @\xi \mid S, \xi@$
$\eta \in \mathbf{Env}$	$:=$	$\emptyset \mid \eta, x \mapsto h$
	$\mid$	$\eta, \alpha \mapsto \tau \mid \eta, \beta \mapsto \varphi \mid \eta, \delta \mapsto \xi$

Figure 1. Syntax of the target language

Proofs of correctness for these analyses need to take these particularities into account. So, to demonstrate the correctness of our sharing analysis, we equip our target language with an operational semantics for call-by-need evaluation that distinguishes between updatable and nonupdatable closures (Section 4.2). Correctness of the analysis then follows from the soundness properties of a typing relation for target terms (Section 4.3) with respect to this particular operational semantics (Section 4.4).

A correctness result for uniqueness typing remains as future work. (Note that the way in which such a result is obtained depends heavily on the way in which we address the issues that arise from the interaction between subeffecting and the containment restriction; see Section 3.4.)

4.1 Syntax

Figure 1 shows the syntax of our target calculus. In the term language we distinguish between variable terms and nonvariable terms. The latter are referred to as *preterms* and are ranged over by the metavariable u. As we will see in Section 4.2, the explicit distinction between variables and preterms merely facilitates the definition of an operational semantics. Preterms subsume function abstractions, two forms of function application (one with a variable in argument position, one with a preterm), abstractions and applications for types, effects, and *evidence* (see below), nonrecursive local definitions, and explicit subeffect coercions. Explicit effect annotations appear on function abstractions, function parameters, preterm arguments, and let-bindings. The set of variables that occur free in a term t is written as $\mathrm{fv}(t)$.

Qualifiers are built from effect expressions and a single inequality symbol $\sqsubseteq$. Predicates of the form $\varphi_1 \sqsupseteq \varphi_2$ are just syntactic sugar for $\varphi_2 \sqsubseteq \varphi_1$. Evidence expressions act as explicit proofs that predicates hold: the constructors $\imath$ and $\circ$ establish, respectively, the reflexivity and transitivity of the ordering on effects, while $\bot$ and $\top$ denote that 1 and ω are, respectively, the least and maximal element of the effect ordering.

In the term language, evidence is used to witness the validity of explicit subeffect coercions $\xi\ t$. For example, the function that applies its first argument to its second (cf. *apply* in Sections 3.3 and 3.4) is, in our target language, encoded by

$$\begin{array}{l}\lambda\mathsf{a}.\ \lambda\mathsf{b}.\\ \quad \lambda\mathsf{p}_1.\ \lambda\mathsf{p}_2.\ \lambda\mathsf{q}.\ \lambda\mathsf{r}_1.\ \lambda\mathsf{r}_2.\\ \quad\quad \lambda p : \mathsf{p}_1 \diamond \mathsf{p}_2.\ \lambda r : \mathsf{r}_1 \sqsupseteq \mathsf{r}_2.\\ \quad\quad\quad \lambda^{\omega} f^{\mathsf{r}_1} : \mathsf{a}^{\mathsf{p}_1} \to \mathsf{b}^{\mathsf{q}}.\ \lambda^{\mathsf{r}_2} x^{\mathsf{p}_2} : \mathsf{a}.\ f\ (p\ x)^{\mathsf{p}_1}.\end{array}$$

Here, the subeffect coercion $p\ x$ uses the evidence argument p to witness the adaptation of the effect p_2 of x to the parameter effect p_1 of the function f.

4.2 Operational Semantics

To capture lazy evaluation in an operational semantics, we employ a variation of Sestoft's simple abstract machine for call-by-need reduction (Sestoft 1997). The main difference between ours and Sestoft's semantics is that the latter limits all function arguments to variables. As a result, closures are created only at let-bindings and simply passed through at lambda-bindings. Although this restriction can be met easily—for instance, by assuming a preprocessor that turns all applications of the form $t\ u^{\varphi}$ into local definitions $\mathbf{let}\ x^{\varphi} = u\ \mathbf{in}\ t\ x$ with $x \notin \mathrm{fv}(t)$—we choose not to include it in our system, allowing closures to be introduced at application sites as well and thus retaining a certain degree of flexibility in the term language. However, to determine whether or not evaluation of an application indeed produces a new closure, we have to be able to tell apart arguments that do give rise to closures from those that do not. This amounts exactly to our distinction between preterms and variables.

The operational semantics of our target language is now defined by means of a reduction relation between abstract-machine configurations. Such configurations consist of a heap H, a control term t, a stack S, and an environment η. Heaps act as finite maps from heap locations h to effect-annotated closures $(t;\ \eta)^{\varphi}$. We write $H(h)$ for the closure associated with the rightmost binding for h in H. The set of all locations bound in H is written as $\mathrm{dom}(H)$, whereas $H \setminus h$ stands for the heap obtained by removing all bindings for h from H. To keep track of evaluation contexts, we maintain a stack of context markers. A marker $\#h$, for example, indicates that the term under evaluation is to be written to the heap at location h. Likewise, a marker $@h$ indicates that the control term is to be applied to the closure stored in h. Other markers denote abstraction and application contexts for types, effects, and evidence, and subeffecting contexts. Environments make beta-substitution explicit by mapping term variables to heap locations, type variables to types, effect variables to effects, and evidence variables to evidence expressions. We write $\eta(x)$ for the location that is associated with the rightmost binding for x in η. In the same fashion, we write $\eta(\alpha)$, $\eta(\beta)$, and $\eta(\delta)$ to retrieve the rightmost bindings for type, effect, and evidence variables.

The reduction rules for the abstract machine are shown in Figure 2. Each rule specifies a single evaluation step. The idea is that the machine takes such steps repeatedly until its stack is empty and its control term has reached weak-head normal form. The reduction rules are syntax-directed: for each possible combination of a control term and a stack, at most one rule applies. Laziness is achieved through interplay between the rules R-VarMany and R-WhnfUpd. The former expresses that, when a shared closure is entered, an update marker is pushed on the stack. The latter prescribes that, when the control term reaches weak-head normal form with an update marker on top of the stack, the heap is updated accordingly. Rule

Reduction $H;\ t;\ S;\ \eta \longrightarrow H';\ t';\ S';\ \eta'$

$$\frac{\eta(x) = h \quad H(h) = (t';\ \eta')^1}{H;\ x;\ S;\ \eta \longrightarrow H \setminus h;\ t';\ S;\ \eta'} \text{(R-VarOnce)}$$

$$\frac{\varphi \in \{\beta, \omega\} \quad \eta(x) = h \quad H(h) = (t';\ \eta')^{\varphi}}{H;\ x;\ S;\ \eta \longrightarrow H;\ t';\ (S, \#h);\ \eta'} \text{(R-VarMany)}$$

$$H;\ \lambda^{\varphi_1} x^{\varphi_2} : \tau.\ t_1;\ (S', @h);\ \eta \longrightarrow H;\ t_1;\ S';\ (\eta, x \mapsto h) \text{(R-AbsApp)}$$

$$\frac{\eta(x) = h}{H;\ t_1\ x;\ S;\ \eta \longrightarrow H;\ t_1;\ (S, @h);\ \eta} \text{(R-AppVar)}$$

$$\frac{h \notin \mathrm{dom}(H)}{H;\ t_1\ u^{\varphi};\ S;\ \eta \longrightarrow (H, h \mapsto (u;\ \eta)^{\varphi});\ t_1;\ (S, @h);\ \eta} \text{(R-AppPreTm)}$$

$$\frac{t_1 \notin \textbf{Whnf}}{H;\ \lambda\alpha.\ t_1;\ S;\ \eta \longrightarrow H;\ t_1;\ (S, \lambda\alpha);\ \eta} \text{(R-TyAbs)}$$

$$H;\ \lambda\alpha.\ w_1;\ (S', @\tau);\ \eta \longrightarrow H;\ w_1;\ S';\ (\eta, \alpha \mapsto \tau) \text{(R-TyAbsTyApp)}$$

$$H;\ t_1\ \tau;\ S;\ \eta \longrightarrow H;\ t_1;\ (S, @\tau);\ \eta \text{(R-TyApp)}$$

$$\frac{t_1 \notin \textbf{Whnf}}{H;\ \lambda\beta.\ t_1;\ S;\ \eta \longrightarrow H;\ t_1;\ (S, \lambda\beta);\ \eta} \text{(R-EffAbs)}$$

$$H;\ \lambda\beta.\ w_1;\ (S', @\varphi);\ \eta \longrightarrow H;\ w_1;\ S';\ (\eta, \beta \mapsto \varphi) \text{(R-EffAbsEffApp)}$$

$$H;\ t_1\ \varphi;\ S;\ \eta \longrightarrow H;\ t_1;\ (S, @\varphi);\ \eta \text{(R-EffApp)}$$

$$\frac{t_1 \notin \textbf{Whnf}}{H;\ \lambda\delta :: \pi.\ t_1;\ S;\ \eta \longrightarrow H;\ t_1;\ (S, \lambda\delta :: \pi);\ \eta} \text{(R-EvAbs)}$$

$$H;\ \lambda\delta :: \pi.\ w_1;\ (S', @\xi);\ \eta \longrightarrow H;\ w_1;\ S';\ (\eta, \delta \mapsto \xi) \text{(R-EvAbsEvApp)}$$

$$H;\ t_1\ \xi;\ S;\ \eta \longrightarrow H;\ t_1;\ (S, @\xi);\ \eta \text{(R-EvApp)}$$

$$\frac{h \notin \mathrm{dom}(H)}{H;\ \textbf{let}\ x^{\varphi} = t_1\ \textbf{in}\ t_2;\ S;\ \eta \longrightarrow (H, h \mapsto (t_1;\ \eta)^{\varphi});\ t_2;\ S;\ (\eta, x \mapsto h)} \text{(R-Let)}$$

$$\frac{t_1 \notin \textbf{Whnf}}{H;\ \xi\ t_1;\ S;\ \eta \longrightarrow H;\ t_1;\ (S, \xi@);\ \eta} \text{(R-Sub)}$$

$$\frac{H(h) = (t', \eta')^{\varphi}}{H;\ w;\ (S', \#h);\ \eta \longrightarrow (H, h \mapsto (w;\ \eta)^{\varphi});\ w;\ S';\ \eta} \text{(R-WhnfUpd)}$$

$$H;\ w;\ (S', \lambda\alpha);\ \eta \longrightarrow H;\ \lambda\alpha.\ w;\ S';\ \eta \text{(R-WhnfTyAbs)}$$

$$H;\ w;\ (S', \lambda\beta);\ \eta \longrightarrow H;\ \lambda\beta.\ w;\ S';\ \eta \text{(R-WhnfEffAbs)}$$

$$H;\ w;\ (S', \lambda\delta : \pi);\ \eta \longrightarrow H;\ \lambda\delta : \pi.\ w;\ S';\ \eta \text{(R-WhnfEvAbs)}$$

$$H;\ w;\ (S', \xi@);\ \eta \longrightarrow H;\ \xi\ w;\ S';\ \eta \text{(R-WhnfSub)}$$

Figure 2. Update avoiding operational semantics for the target language

R-VarOnce says that, after they are entered, nonupdatable closures are immediately removed from the heap.

Further examination of the reduction rules reveals that types and evidence do not have any operational significance. They are merely there to guide type checking (see Section 4.3) and to facilitate proofs. The same holds for subeffect coercions. Also, effect abstractions and effect applications play no decisive rôle in the evaluation of terms. The only effect expressions that do influence the operation of our abstract machine are those that appear on preterm arguments and let-bindings. These are the effects that are used as heap annotations and thus drive the decision whether or not entered closures are updated. However, note that we are careful not to subject the effects on closures to beta-substitution: this would cause effect abstractions and applications to have nontrivial operational meaning after all. Instead, whenever we encounter an effect variable as the annotation on an entered closure, we conservatively assume the closure is to be updated; see rule R-VarMany.

As a result, implementations can safely employ an *erasure semantics* and strip off operationally insignificant abstractions, applications, and coercions, before actually interpreting a program or generating code for it. This way, we assure that our usage analysis does not impose the unnecessary run-time overhead associated with the evaluation of types, effects, and evidence.

4.3 Typing

To be able to statically ensure that a given program in the target language does not "go wrong", we employ a type and effect system that specifies which target terms are well-typed with respect to a context Γ. Such contexts map term variables to types and effects (through entries of the form $x :^{\varphi} \tau$), and evidence variables to qualifiers (through entries of the form $\delta : \pi$). We write $\Gamma(x)$ for the rightmost binding for x in Γ and, similarly, $\Gamma(\delta)$ for the rightmost binding for δ in Γ. The set of type variables that occur free in Γ is written as $\mathrm{ftv}(\Gamma)$; the set of free effect variables in Γ is denoted by $\mathrm{fev}(\Gamma)$. We write $\Gamma \setminus x$ for the environment that is obtained by removing all bindings for x from Γ.

The effect typing relation is specified in Figure 3 through judgements of the form $\Gamma \vdash t :^{\varphi} \tau$, indicating that, in context Γ, term t can be assigned type τ and effect φ.

When typing terms that have two or more subterms, i.e., function applications (T-AppVar and T-AppPreTm) and local definitions (T-Let), we split up all *potentially* unique variables that appear in the context (so, these include those that are bound to a variable effect), and distribute them over the contexts that are passed down to the subterms. This splitting of contexts is described by rules of the form $\Gamma = \Gamma_1 \bowtie \Gamma_2$. Our use of context splitting ensures that unique variables are indeed used at most once along every possible path in a program's control-flow graph.

The well-formedness of evidence expressions is established by a set of subsidiary rules of the form $\Gamma \vdash \xi : \pi$, indicating that, in context Γ, evidence ξ is a valid proof of predicate π. These rules are invoked in T-Abs, to meet the containment restriction for function abstractions, and in T-Sub, to enforce the validity of evidence expressions that witness explicit subeffect coercions. Note how T-Sub makes use of the system parameter $\diamond$, that is (cf. Section 3.1) instantiated to either $\sqsubseteq$ (for sharing analysis) or $\sqsupseteq$ (for uniqueness typing).

4.4 Soundness

To demonstrate the soundness of the type and effect system, we extend the typing relation on terms to a typing relation on full programs, i.e., abstract-machine configurations. The resulting relation is defined by a set of rules of the form $H;\ \eta;\ S \vdash t :^{\varphi} \tau$ and is shown in Figure 4.

Intuitively, the rules for program typing unwind the stack to reconstruct the evaluation context. The type of a program is then established in a typing context that is obtained through an auxiliary judgement $H;\ \eta \vdash \Gamma$. This auxiliary judgement combines the sub-

Context splitting $\boxed{\Gamma = \Gamma_1 \bowtie \Gamma_2}$

$$\emptyset = \emptyset \bowtie \emptyset \quad \text{(C-Empty)}$$

$$\frac{\varphi \in \{\beta, 1\} \quad \Gamma_1 = \Gamma_{11} \bowtie \Gamma_{12}}{\Gamma_1, x :^{\varphi} \tau = \Gamma_{11}, x :^{\varphi} \tau \bowtie \Gamma_{12} \setminus x} \quad \text{(C-VarOnce1)}$$

$$\frac{\varphi \in \{\beta, 1\} \quad \Gamma_1 = \Gamma_{11} \bowtie \Gamma_{12}}{\Gamma_1, x :^{\varphi} \tau = \Gamma_{11} \setminus x \bowtie \Gamma_{12}, x :^{\varphi} \tau} \quad \text{(C-VarOnce2)}$$

$$\frac{\Gamma_1 = \Gamma_{11} \bowtie \Gamma_{12}}{\Gamma_1, x :^{\omega} \tau = \Gamma_{11}, x :^{\omega} \tau \bowtie \Gamma_{12}, x :^{\omega} \tau} \quad \text{(C-VarMany)}$$

$$\frac{\Gamma_1 = \Gamma_{11} \bowtie \Gamma_{12}}{\Gamma_1, \delta : \pi = \Gamma_{11}, \delta : \pi \bowtie \Gamma_{12}, \delta : \pi} \quad \text{(C-EvVar)}$$

Evidence typing $\boxed{\Gamma \vdash \xi : \pi}$

$$\frac{\Gamma(\delta) = \pi}{\Gamma \vdash \delta : \pi} \quad \text{(Q-Var)}$$

$$\Gamma \vdash \imath : \varphi \sqsubseteq \varphi \quad \text{(Q-Refl)}$$

$$\frac{\Gamma \vdash \xi_2 : \varphi_1 \sqsubseteq \varphi_2 \quad \Gamma \vdash \xi_1 : \varphi_2 \sqsubseteq \varphi_3}{\Gamma \vdash \xi_1 \circ \xi_2 : \varphi_1 \sqsubseteq \varphi_3} \quad \text{(Q-Trans)}$$

$$\Gamma \vdash \bot : 1 \sqsubseteq \varphi \quad \text{(Q-Bot)}$$

$$\Gamma \vdash \top : \varphi \sqsubseteq \omega \quad \text{(Q-Top)}$$

Term typing $\boxed{\Gamma \vdash t :^{\varphi} \tau}$

$$\frac{\Gamma(x) =^{\varphi} \tau}{\Gamma \vdash x :^{\varphi} \tau} \quad \text{(T-Var)}$$

$$\frac{\mathrm{fv}(t_1) - \{x\} = \{x_1, ..., x_n\} \quad \left.\begin{array}{l} \Gamma(x_i) =^{\varphi_{x_i}} \tau_{x_i} \\ \Gamma \vdash \xi_i : \varphi \sqsubseteq \varphi_{x_i} \end{array}\right\} \text{for each } i \in \{1, ..., n\} \quad \Gamma, x :^{\varphi_1} \tau_1 \vdash t_1 :^{\varphi_2} \tau_2}{\Gamma \vdash (\lambda^{\varphi} x^{\varphi_1} : \tau_1.\ t_1) :^{\varphi} \tau_1^{\varphi_1} \to \tau_2^{\varphi_2}} \quad \text{(T-Abs)}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1 \vdash t_1 :^{\varphi_1} \tau_2^{\varphi_2} \to \tau^{\varphi} \quad \Gamma_2 \vdash x :^{\varphi_2} \tau_2}{\Gamma \vdash t_1\ x :^{\varphi} \tau} \quad \text{(T-AppVar)}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1 \vdash t_1 :^{\varphi_1} \tau_2^{\varphi_2} \to \tau^{\varphi} \quad \Gamma_2 \vdash u :^{\varphi_2} \tau_2}{\Gamma \vdash t_1\ u^{\varphi_2} :^{\varphi} \tau} \quad \text{(T-AppPreTm)}$$

$$\frac{\alpha \notin \mathrm{ftv}(\Gamma) \quad \Gamma \vdash t_1 :^{\varphi} \tau_1}{\Gamma \vdash \lambda\alpha.\ t_1 :^{\varphi} \forall\alpha.\ \tau_1} \quad \text{(T-TyAbs)}$$

$$\frac{\Gamma \vdash t_1 :^{\varphi} \forall\alpha.\ \tau_1}{\Gamma \vdash t_1\ \tau_2 :^{\varphi} [\alpha \mapsto \tau_2]\tau_1} \quad \text{(T-TyApp)}$$

$$\frac{\beta \notin \mathrm{fev}(\Gamma) \cup \{\varphi\} \quad \Gamma \vdash t_1 :^{\varphi} \tau_1}{\Gamma \vdash \lambda\beta.\ t_1 :^{\varphi} \forall\beta.\ \tau_1} \quad \text{(T-EffAbs)}$$

$$\frac{\Gamma \vdash t_1 :^{\varphi} \forall\beta.\ \tau_1}{\Gamma \vdash t_1\ \varphi_2 :^{\varphi} [\beta \mapsto \varphi_2]\tau_1} \quad \text{(T-EffApp)}$$

$$\frac{\Gamma, \delta : \pi \vdash t_1 :^{\varphi} \tau_1}{\Gamma \vdash (\lambda\delta : \pi.\ t_1) :^{\varphi} \pi \Rightarrow \tau_1} \quad \text{(T-EvAbs)}$$

$$\frac{\Gamma \vdash t_1 :^{\varphi} \pi \Rightarrow \tau_1 \quad \Gamma \vdash \xi : \pi}{\Gamma \vdash t_1\ \xi :^{\varphi} \tau_1} \quad \text{(T-EvApp)}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1 \vdash t_1 :^{\varphi_1} \tau_1 \quad \Gamma_2, x :^{\varphi_1} \tau_1 \vdash t_2 :^{\varphi} \tau}{\Gamma \vdash \mathbf{let}\ x^{\varphi_1} = t_1\ \mathbf{in}\ t_2 :^{\varphi} \tau} \quad \text{(T-Let)}$$

$$\frac{\Gamma \vdash t_1 :^{\varphi_1} \tau \quad \Gamma \vdash \xi : \varphi \diamond \varphi_1}{\Gamma \vdash \xi\ t_1 :^{\varphi} \tau} \quad \text{(T-Sub)}$$

Figure 3. Type and effect system of the target language

Store typing $\boxed{H;\ \eta \vdash \Gamma}$

$$H;\ \emptyset \vdash \emptyset \quad \text{(S-Empty)}$$

$$\frac{H;\ \eta_1 \vdash \Gamma_1 \quad H(h) = (t;\ \eta')^{\varphi} \quad H;\ \eta' \vdash \Gamma' \quad \Gamma' \vdash t :^{\varphi} \tau}{H;\ \eta_1, x \mapsto h \vdash \Gamma_1, x :^{\varphi} \tau} \quad \text{(S-Var)}$$

$$\frac{H;\ \eta_1 \vdash \Gamma}{H;\ \eta_1, \alpha \mapsto \tau \vdash \Gamma} \quad \text{(S-TyVar)}$$

$$\frac{H;\ \eta_1 \vdash \Gamma}{H;\ \eta_1, \beta \mapsto \varphi \vdash \Gamma} \quad \text{(S-EffVar)}$$

$$\frac{H;\ \eta_1 \vdash \Gamma_1 \quad \Gamma_1 \vdash \xi : \pi}{H;\ \eta_1, \delta \mapsto \xi \vdash \Gamma_1, \delta : \pi} \quad \text{(S-EvVar)}$$

Program typing $\boxed{H;\ \eta;\ S \vdash t :^{\varphi} \tau}$

$$\frac{H;\ \eta \vdash \Gamma \quad \Gamma \vdash t :^{\varphi'} \tau' \quad \eta(\varphi') = \varphi \quad \eta(\tau') = \tau}{H;\ \eta;\ \emptyset \vdash t :^{\varphi} \tau} \quad \text{(P-Empty)}$$

$$\frac{H(h) = (t', \eta')^{\varphi} \quad H;\ \eta';\ S_1 \vdash t' :^{\varphi} \tau \quad H;\ \eta;\ S_1 \vdash t :^{\varphi} \tau}{H;\ \eta;\ S_1, \#h \vdash t :^{\varphi} \tau} \quad \text{(P-Upd)}$$

$$\frac{H(h) = (t', \eta')^{\varphi'} \quad H;\ \eta';\ \emptyset \vdash t' :^{\varphi'} \tau' \quad H;\ \eta;\ S_1 \vdash t :^{\varphi_1} \tau'^{\varphi'} \to \tau^{\varphi}}{H;\ \eta;\ S_1, @h \vdash t :^{\varphi} \tau} \quad \text{(P-App)}$$

$$\frac{H;\ \eta;\ S_1 \vdash \lambda\alpha.\ t :^{\varphi} \tau}{H;\ \eta;\ S_1, \lambda\alpha \vdash t :^{\varphi} \tau} \quad \text{(P-TyAbs)}$$

$$\frac{H;\ \eta;\ S_1 \vdash t\ \tau' :^{\varphi} \tau}{H;\ \eta;\ S_1, @\tau' \vdash t :^{\varphi} \tau} \quad \text{(P-TyApp)}$$

$$\frac{H;\ \eta;\ S_1 \vdash \lambda\beta.\ t :^{\varphi} \tau}{H;\ \eta;\ S_1, \lambda\beta \vdash t :^{\varphi} \tau} \quad \text{(P-EffAbs)}$$

$$\frac{H;\ \eta;\ S_1 \vdash t\ \varphi' :^{\varphi} \tau}{H;\ \eta;\ S_1, @\varphi' \vdash t :^{\varphi} \tau} \quad \text{(P-EffApp)}$$

$$\frac{H;\ \eta;\ S_1 \vdash \lambda\delta : \pi.\ t :^{\varphi} \tau}{H;\ \eta;\ S_1, \lambda\delta : \pi \vdash t :^{\varphi} \tau} \quad \text{(P-EvAbs)}$$

$$\frac{H;\ \eta;\ S_1 \vdash t\ \xi :^{\varphi} \tau}{H;\ \eta;\ S_1, @\xi \vdash t :^{\varphi} \tau} \quad \text{(P-EvApp)}$$

$$\frac{H;\ \eta;\ S_1 \vdash \xi\ t :^{\varphi} \tau}{H;\ \eta;\ S_1, \xi@ \vdash t :^{\varphi} \tau} \quad \text{(P-Sub)}$$

Figure 4. Program typing

stitutions that are recorded in the program's environment with the types of the terms on the heap to produce a compliant typing context. In rule P-EMPTY, we write $\eta(\varphi)$ and $\eta(\tau)$ for the application of the substitution recorded in η to, respectively, an effect φ and a type τ.

An important point to make is that the well-typedness of a program is not enough to guarantee that the abstract machine does not get "stuck". As pointed out in Section 4.2, evidence abstractions and evidence applications play no rôle in controlling the operation of the abstract machine. This enables the definition of a sound erasure semantics, but also forces us to be careful when establishing the correctness of our sharing analysis. For instance, if we consistently extend our target calculus with integer constants, the following program is well-typed,

$$\begin{array}{l}\lambda p : \omega \sqsubseteq 1.\\ \quad \textbf{let } \mathit{twice}^1 = \lambda^1 f^\omega : \mathsf{Int}^1 \to \mathsf{Int}^1.\, \lambda^1 x^1 : \mathsf{Int}.\, f\ (f\ x)^1\\ \quad \textbf{in } \ \mathit{twice}\ (p\ (\lambda^1 y^1 : \mathsf{Int}.\, y))^\omega\ 2,\end{array}$$

but nevertheless fails hopelessly when run on our abstract machine. To see why, consider the top-level evidence abstraction. It indicates that the program expects evidence p for the predicate $\omega \sqsubseteq 1$. Obviously, our evidence typing rules preclude the validity of any such evidence, but in the body of the evidence abstraction, we can nevertheless assume that the evidence is provided and, so, we use it here to lift the effect of the nonupdatable abstraction $\lambda^1 y^1 : \mathsf{Int}.\, y$. However, evidence and type abstractions have no real operational meaning, so machine reduction effectively ignores the top-level lambdas and immediately starts the evaluation of the local definition. Inside the local definition, the program behaves as if evidence for $\omega \sqsubseteq 1$ was indeed provided, and the machine gets stuck inside the body of *twice*, when it tries to enter the closure for the abstraction $\lambda^1 y^1 : \mathsf{Int}.\, y$ for the second time and finds it removed from the heap.

So, to demonstrate the correctness of our sharing analysis, we need a stronger condition than mere well-typedness of target programs. To this end, we introduce a notion of *satisfiability*, that prescribes that instances of polymorphic types leave the ordering on effects antisymmetric:

Definition 1.

1. A typing context Γ is *consistent* if there does not exist evidence ξ, such that $\Gamma \vdash \xi : \omega \sqsubseteq 1$.
2. A type τ is *satisfiable* in Γ if (a) $\tau = \alpha$, or (b) $\tau = \tau_1^{\varphi_1} \to \tau_2^{\varphi_2}$ with τ_1 and τ_2 satisfiable in Γ, or (c) $\tau = \forall\alpha.\ \tau_1$ with τ_1 satisfiable in Γ, or (d) $\tau = \forall\beta.\ \tau_1$ with τ_1 satisfiable in Γ, or (e) $\tau = \pi \Rightarrow \tau_1$ with, for all δ, $(\Gamma, \delta : \pi)$ consistent and τ_1 satisfiable in $(\Gamma, \delta : \pi)$. ∎

With satisfiability in place, soundness for the target language with respect to the operational semantics of Figure 2 is established, as usual, by theorems for progress and preservation:

Theorem 2 (Progress). Let $\diamond = \sqsubseteq$. If $H;\ \eta;\ S \vdash t :^\varphi \tau$, then either (a) $t \in \mathbf{Whnf}$ and $S = \emptyset$, or else (b) the existence of an h with $h \notin \mathrm{dom}(H)$ implies that $H;\ t;\ S;\ \eta \longrightarrow H';\ t';\ S';\ \eta'$ for some H', t', S' and η'. ∎

Theorem 3 (Preservation). Let $\diamond = \sqsubseteq$ and $H;\ \eta;\ S \vdash t :^\varphi \tau$ with $H;\ \eta \vdash \Gamma$, such that Γ is consistent and τ satisfiable in Γ. If $H;\ t;\ S;\ \eta \longrightarrow H';\ t';\ S';\ \eta'$, then (a) $H';\ \eta';\ S' \vdash t' :^\varphi \tau$, and (b) there exists a consistent Γ' with $H';\ \eta' \vdash \Gamma'$ and τ satisfiable in Γ'. ∎

5. Effect Reconstruction

Of course, we do not expect programmers to write programs in our target language: getting all explicit annotations right is tedious and error-prone. Instead, we rely on an algorithm that analyses an unannotated source program and then decorates it with the appropriate annotations. Developing and implementing such an algorithm is far from trivial, but in our approach we can fortunately reuse the machinery that is already available within the framework of qualified types.

In this section, we introduce an implicitly typed source language (Section 5.1) and discuss its translation into the target language (Section 5.2). The translation is driven by derivations in a type and effect system for source terms. This type and effect system, which is a conservative extension of the Hindley-Milner system (Section 5.4), has a principal-type property, that drives the development of an incremental algorithm for effect reconstruction (Section 5.3).

Term language

$$\widehat{t} \in \widehat{\mathbf{Tm}} \quad ::= \quad x \mid \lambda x.\, \widehat{t} \mid \widehat{t}\,\widehat{t} \mid \textbf{let } x = \widehat{t} \textbf{ in } \widehat{t}$$

Type and effect language

$$\begin{array}{lll}\widehat{\tau} \in \widehat{\mathbf{Ty}} & ::= & \alpha \mid \widehat{\tau}^{\varphi} \to \widehat{\tau}^{\varphi}\\ \widehat{\rho} \in \widehat{\mathbf{QualTy}} & ::= & \widehat{\tau} \mid \pi \Rightarrow \widehat{\rho}\\ \widehat{\sigma} \in \widehat{\mathbf{TyScheme}} & ::= & \widehat{\rho} \mid \forall\alpha.\, \widehat{\sigma} \mid \forall\beta.\, \widehat{\sigma}\end{array}$$

Typing contexts

$$\widehat{\Gamma} \in \widehat{\mathbf{Ctx}} \quad ::= \quad \emptyset \mid \widehat{\Gamma}, x :^{\varphi} \widehat{\sigma} \mid \widehat{\Gamma}, \pi$$

Figure 5. Syntax of the source language

5.1 Source Language

The surface language of our system is an implicitly typed, let-polymorphic lambda-calculus with a Hindley-Milner-like type system (Milner 1978). Its syntax is given in Figure 5. To distinguish them from their counterparts in the target language, we decorate metavariables and nonterminals of the source language with "hats", as in $\widehat{t}$ and $\widehat{\mathbf{Tm}}$,

The typing rules for source terms are, together with auxiliary rules for context splitting and predicate entailment, presented in Figure 6. Typing contexts are essentially the same as those in the target language with the notable exception that predicates are not associated with evidence variables. The type language is stratified into monomorphic types, qualified types, and type schemes. We write $\mathrm{fev}(\widehat{\sigma})$ for the effect variables that appear free in a type scheme $\widehat{\sigma}$. In contrast to the typing rules for the target language, the typing rules in Figure 6 are not syntax-directed.

5.2 Translation

In a way, the typing rules for the source language make up the actual usage analysis. Provided with a type derivation for a term $\widehat{t}$, we can systematically produce an annotated type t, such that t is a *completion* of $\widehat{t}$:

Definition 4.

1. A source term $\widehat{t}$ is *completed* by a target term t if and only if $\lfloor t \rfloor = \widehat{t}$, where $\lfloor t \rfloor$ is given by:

$$\begin{array}{ll}\lfloor x \rfloor & = x\\ \lfloor \lambda^{\varphi_1} x^{\varphi_2} : \tau.\, t_1 \rfloor & = \lambda x.\, \lfloor t_1 \rfloor\\ \lfloor t_1\ x \rfloor & = \lfloor t_1 \rfloor\ x\\ \lfloor t_1\ u^{\varphi} \rfloor & = \lfloor t_1 \rfloor\ \lfloor u \rfloor\\ \lfloor \lambda\alpha.\, t_1 \rfloor & = \lfloor t_1 \rfloor\\ \lfloor t_1\ \tau \rfloor & = \lfloor t_1 \rfloor\\ \lfloor \lambda\beta.\, t_1 \rfloor & = \lfloor t_1 \rfloor\\ \lfloor t_1\ \varphi \rfloor & = \lfloor t_1 \rfloor\end{array}$$

Context splitting $\boxed{\widehat{\Gamma} = \widehat{\Gamma}_1 \bowtie \widehat{\Gamma}_2}$

$$\emptyset = \emptyset \bowtie \emptyset \quad \text{(CS-Empty)}$$

$$\frac{\varphi \in \{\beta, 1\} \quad \widehat{\Gamma}_1 = \widehat{\Gamma}_{11} \bowtie \widehat{\Gamma}_{12}}{\widehat{\Gamma}_1, x :^{\varphi} \widehat{\sigma} = \widehat{\Gamma}_{11}, x :^{\varphi} \widehat{\sigma} \bowtie \widehat{\Gamma}_{12} \setminus x} \quad \text{(CS-VarOnce1)}$$

$$\frac{\varphi \in \{\beta, 1\} \quad \widehat{\Gamma}_1 = \widehat{\Gamma}_{11} \bowtie \widehat{\Gamma}_{12}}{\widehat{\Gamma}_1, x :^{\varphi} \widehat{\sigma} = \widehat{\Gamma}_{11} \setminus x \bowtie \widehat{\Gamma}_{12}, x :^{\varphi} \widehat{\sigma}} \quad \text{(CS-VarOnce2)}$$

$$\frac{\widehat{\Gamma}_1 = \widehat{\Gamma}_{11} \bowtie \widehat{\Gamma}_{12}}{\widehat{\Gamma}_1, x :^{\omega} \widehat{\sigma} = \widehat{\Gamma}_{11}, x :^{\omega} \widehat{\sigma} \bowtie \widehat{\Gamma}_{12}, x :^{\omega} \widehat{\sigma}} \quad \text{(CS-VarMany)}$$

$$\frac{\widehat{\Gamma}_1 = \widehat{\Gamma}_{11} \bowtie \widehat{\Gamma}_{12}}{\widehat{\Gamma}_1, \pi = \widehat{\Gamma}_{11}, \pi \bowtie \widehat{\Gamma}_{12}, \pi} \quad \text{(CS-Qual)}$$

Predicate entailment $\boxed{\widehat{\Gamma} \Vdash \pi}$

$$\frac{\pi \in \widehat{\Gamma}}{\widehat{\Gamma} \Vdash \pi} \quad \text{(E-Mono)}$$

$$\widehat{\Gamma} \Vdash \varphi \sqsubseteq \varphi \quad \text{(E-Refl)}$$

$$\frac{\widehat{\Gamma} \Vdash \varphi_1 \sqsubseteq \varphi_2 \quad \widehat{\Gamma} \Vdash \varphi_2 \sqsubseteq \varphi_3}{\widehat{\Gamma} \Vdash \varphi_1 \sqsubseteq \varphi_3} \quad \text{(E-Trans)}$$

$$\widehat{\Gamma} \Vdash 1 \sqsubseteq \varphi \quad \text{(E-Bot)}$$

$$\widehat{\Gamma} \Vdash \varphi \sqsubseteq \omega \quad \text{(E-Top)}$$

Term typing $\boxed{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \widehat{\sigma}}$

$$\frac{\widehat{\Gamma}(x) =^{\varphi} \widehat{\sigma}}{\widehat{\Gamma} \vdash x :^{\varphi} \widehat{\sigma}} \quad \text{(UA-Var)}$$

$$\frac{\mathrm{fv}(\widehat{t}_1) - \{x\} = \{x_1, ..., x_n\} \quad \left.\begin{array}{l}\widehat{\Gamma}(x_i) =^{\varphi_{x_i}} \widehat{\sigma}_{x_i} \\ \widehat{\Gamma} \vdash \varphi \sqsubseteq \varphi_{x_i}\end{array}\right\} \text{for each } i \in \{1, ..., n\} \quad \widehat{\Gamma}, x :^{\varphi_1} \widehat{\tau}_1 \vdash \widehat{t}_1 :^{\varphi_2} \widehat{\tau}_2}{\widehat{\Gamma} \vdash \lambda x.\, \widehat{t}_1 :^{\varphi} \widehat{\tau}_1^{\varphi_1} \to \widehat{\tau}_2^{\varphi_2}} \quad \text{(UA-Abs)}$$

$$\frac{\widehat{\Gamma} = \widehat{\Gamma}_1 \bowtie \widehat{\Gamma}_2 \quad \widehat{\Gamma}_1 \vdash \widehat{t}_1 :^{\varphi_1} \widehat{\tau}_2^{\varphi_2} \to \widehat{\tau}^{\varphi} \quad \widehat{\Gamma}_2 \vdash \widehat{t}_2 :^{\varphi_2} \widehat{\tau}_2}{\widehat{\Gamma} \vdash \widehat{t}_1\, \widehat{t}_2 :^{\varphi} \widehat{\tau}} \quad \text{(UA-App)}$$

$$\frac{\widehat{\Gamma} = \widehat{\Gamma}_1 \bowtie \widehat{\Gamma}_2 \quad \widehat{\Gamma}_1 \vdash \widehat{t}_1 :^{\varphi_1} \widehat{\sigma}_1 \quad \widehat{\Gamma}_2, x :^{\varphi_1} \widehat{\sigma}_1 \vdash \widehat{t}_2 :^{\varphi} \widehat{\sigma}}{\widehat{\Gamma} \vdash \textbf{let } x = \widehat{t}_1 \textbf{ in } \widehat{t}_2 :^{\varphi} \widehat{\sigma}} \quad \text{(UA-Let)}$$

$$\frac{\widehat{\Gamma}, \pi \vdash \widehat{t} :^{\varphi} \widehat{\rho}_1}{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \pi \Rightarrow \widehat{\rho}_1} \quad \text{(UA-Qual)}$$

$$\frac{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \pi \Rightarrow \widehat{\rho} \quad \widehat{\Gamma} \Vdash \pi}{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \widehat{\rho}} \quad \text{(UA-Res)}$$

$$\frac{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \widehat{\sigma}_1 \quad \alpha \notin \mathrm{ftv}(\widehat{\Gamma})}{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \forall \alpha.\, \widehat{\sigma}_1} \quad \text{(UA-TyGen)}$$

$$\frac{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \forall \alpha.\, \widehat{\sigma}_1}{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} [\alpha \mapsto \widehat{\tau}]\widehat{\sigma}_1} \quad \text{(UA-TyInst)}$$

$$\frac{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \widehat{\sigma}_1 \quad \beta \notin \mathrm{fev}(\widehat{\Gamma})}{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \forall \beta.\, \widehat{\sigma}_1} \quad \text{(UA-EffGen)}$$

$$\frac{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \forall \beta.\, \widehat{\sigma}_1}{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} [\beta \mapsto \varphi']\widehat{\sigma}_1} \quad \text{(UA-EffInst)}$$

$$\frac{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi'} \widehat{\sigma} \quad \widehat{\Gamma} \vdash \varphi \diamond \varphi'}{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \widehat{\sigma}} \quad \text{(UA-Sub)}$$

Figure 6. Generic usage analysis for the source language

$$\begin{array}{ll} \lfloor \lambda \delta : \pi.\, t_1 \rfloor & = \lfloor t_1 \rfloor \\ \lfloor t_1\ \xi \rfloor & = \lfloor t_1 \rfloor \\ \lfloor \textbf{let } x^{\varphi} = t_1 \textbf{ in } t_2 \rfloor & = \textbf{let } x = \lfloor t_1 \rfloor \textbf{ in } \lfloor t_2 \rfloor \\ \lfloor \xi\ t_1 \rfloor & = \lfloor t_1 \rfloor. \end{array}$$

2. A source context $\widehat{\Gamma}$ is completed by a target context Γ, if, and only if, $\lfloor \Gamma \rfloor = \widehat{\Gamma}$, where $\lfloor \Gamma \rfloor$ is given by:

$$\begin{array}{ll} \lfloor \emptyset \rfloor & = \emptyset \\ \lfloor \Gamma_1, x :^{\varphi} \tau \rfloor & = \lfloor \Gamma_1 \rfloor, x :^{\varphi} \lfloor \tau \rfloor \\ \lfloor \Gamma_1, \delta : \pi \rfloor & = \lfloor \Gamma_1 \rfloor, \pi. \end{array}$$

■

To systematically obtain completions from well-typed source terms, we define a translation from type derivations in the source language into target terms. As a typical example of a rule in this type-driven translation scheme, consider

$$\left[\!\!\left[\frac{\nabla :: \widehat{\Gamma}, \pi \vdash \widehat{t} :^{\varphi} \widehat{\rho}_1}{\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \pi \Rightarrow \widehat{\rho}_1}\ \text{(UA-Qual)} \right]\!\!\right]_{\Delta} = \lambda \delta : \pi.\, [\![\nabla]\!]_{\Delta, \pi \mapsto \delta} \quad \text{where } \delta \notin \mathrm{cod}(\Delta).$$

Here, ∇ ranges over derivations of the judgement $\widehat{\Gamma}, \pi \vdash \widehat{t} :^{\varphi} \widehat{\rho}_1$. The parameter Δ represents a mapping from predicates to evidence variables. Given such a mapping, we can turn derivations of predicate entailments into evidence, as in

$$\left[\!\!\left[\frac{\pi \in \widehat{\Gamma}}{\widehat{\Gamma} \Vdash \pi}\ \text{(E-Mono)} \right]\!\!\right]_{\Delta} = \Delta(\pi).$$

For reasons of space, we do not present the entire translation here, but the remaining rules are unsurprising and straightforward.

Of course, we are only interested in well-typed completions of source terms. Hence, the translation necessarily preserves well-typedness. As we have seen in Section 4.4, it is also crucial for the correctness of our analyses that the types of target terms are satisfiable. We therefore impose a satisfiability restriction on the type schemes assigned to source terms (cf. Definition 1), and then show that the translation into target terms preserves types and maintains satisfiability:

Definition 5.

1. A typing context $\widehat{\Gamma}$ is *consistent* if $\widehat{\Gamma} \Vdash \omega \sqsubseteq 1$ is not derivable.

2. A type scheme $\widehat{\sigma}$ is *satisfiable* in $\widehat{\Gamma}$ if (a) $\widehat{\sigma} = \widehat{\tau}$, or (b) $\widehat{\sigma} = \pi \Rightarrow \widehat{\rho}_1$ with $(\widehat{\Gamma}, \pi)$ consistent and $\widehat{\rho}_1$ satisfiable in $(\widehat{\Gamma}, \pi)$, or (c) $\widehat{\sigma} = \forall \alpha.\, \widehat{\sigma}_1$ with $\widehat{\sigma}_1$ satisfiable in $\widehat{\Gamma}$, or (d) $\widehat{\sigma} = \forall \beta.\, \widehat{\sigma}_1$ with $\widehat{\sigma}_1$ satisfiable in $\widehat{\Gamma}$. ■

Theorem 6 (Soundness of the Translation). Let $\widehat{\Gamma}$ be consistent and ∇ a derivation of $\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \widehat{\sigma}$ with $\widehat{\sigma}$ satisfiable in $\widehat{\Gamma}$. Then, for $[\![\nabla]\!]_{\Delta} = t$, $\lfloor t \rfloor = \widehat{t}$, $\Gamma \vdash t :^{\varphi} \widehat{\sigma}$, and $\widehat{\sigma}$ satisfiable in Γ, for each Γ with $\lfloor \Gamma \rfloor = \widehat{\Gamma}$ and $\Gamma(\Delta(\pi)) = \pi$ for all $\pi \in \mathrm{dom}(\Delta)$. ■

5.3 Principal Type Schemes

An implementation of our usage analysis is only useful if it uses an algorithm that produces completions corresponding to the "best" assignable type and effect for a given source term. Recall from Section 3.5 that, in our system, "best" does not necessarily mean "least restrictive". Instead, we are interested in the most general type. When a term is used, this type can then be instantiated with the most desirable usage properties permitted by the context. Implementations are thus required to produce completions that correspond to (derivations of) the so-called *principal type scheme* (Hindley 1969) of a source term.

A principal type scheme for a term $\widehat{t}$ is a type scheme that is assignable to $\widehat{t}$ and that is more general than all other assignable type schemes:

Definition 7. A type scheme $\widehat{\sigma}$ is a *solution* for $\widehat{t}$ in $\widehat{\Gamma}$, if $\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \widehat{\sigma}$ for some effect φ. ∎

Definition 8.

1. A type $\widehat{\tau}$ is a *generic instance* of $\widehat{\sigma}$ in $\widehat{\Gamma}$, if $\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \widehat{\sigma}$ implies $\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \widehat{\tau}$.
2. A type scheme $\widehat{\sigma}$ is *more general* than $\widehat{\sigma}'$ in $\widehat{\Gamma}$, if every generic instance of $\widehat{\sigma}'$ in $\widehat{\Gamma}$ is also a generic instance of $\widehat{\sigma}$ in $\widehat{\Gamma}$. ∎

Not all instantiations of a type scheme for a term depend on the context in which the term is used. Consider, for instance, the term $(\lambda x.\ \lambda y.\ x)\ (\lambda x.\ x)$ and its principal type scheme

$$\forall \mathsf{a}.\ \forall \mathsf{p}.\ \forall \mathsf{p}_1.\ \forall \mathsf{p}_2.\ \forall \mathsf{q}_1.\ \forall \mathsf{q}_2.\ \forall \mathsf{r}.\\ \mathsf{p} \sqsubseteq \mathsf{p}_1 \Rightarrow \mathsf{p}_1 \diamond \mathsf{p}_2 \Rightarrow \mathsf{q}_1 \diamond \mathsf{q}_2 \Rightarrow \mathsf{a}^{\mathsf{r}} \to \mathsf{a}^{\mathsf{r}}.$$

The predicates in this type scheme arise from the containment restriction for abstractions and the use of subeffecting at application sites. However, how the effect variables p, p_1, p_2, q_1, and q_2 are instantiated does not matter to any specific use of the term; the only effect variables that do matter are those that appear in the "$\widehat{\tau}$-part" of a type scheme. So, in our example, the only relevant effect variable is r.

Effect variables that do not appear in the $\widehat{\tau}$-part of a type scheme are called *ambiguous* (cf. Jones 1994, Section 5.8) and we do not allow them to appear in inferred type schemes. Instead, our sharing analysis seizes on the opportunity and disambiguates ambiguous variables by instantiating them with their most desirable effect value. In our example, we then infer the type scheme $\forall \mathsf{a}.\ \forall \mathsf{r}.\ \mathsf{a}^{\mathsf{r}} \to \mathsf{a}^{\mathsf{r}}$, and have the function arguments and the partial application annotated with 1. Note that our approach to disambiguation to a large extent coincides with the use of *defaulting* in Haskell (Peyton Jones 2003, Section 4.3.4).

Definition 9.

1. The set of *active effect variables* of a type scheme $\widehat{\sigma}$, written $\mathrm{aev}(\widehat{\sigma})$, is given by
$$\begin{aligned}\mathrm{aev}(\widehat{\tau}) &= \mathrm{fev}(\widehat{\tau})\\ \mathrm{aev}(\pi \Rightarrow \widehat{\rho}_1) &= \mathrm{aev}(\widehat{\rho}_1)\\ \mathrm{aev}(\forall \alpha.\ \widehat{\sigma}_1) &= \mathrm{aev}(\widehat{\sigma}_1)\\ \mathrm{aev}(\forall \beta.\ \widehat{\sigma}_1) &= \mathrm{aev}(\widehat{\sigma}_1).\end{aligned}$$
2. The set of *generic effect variables* of a type scheme $\widehat{\sigma}$, written $\mathrm{gev}(\widehat{\sigma})$, is given by
$$\begin{aligned}\mathrm{gev}(\widehat{\tau}) &= \emptyset\\ \mathrm{gev}(\pi \Rightarrow \widehat{\rho}_1) &= \emptyset\\ \mathrm{gev}(\forall \alpha.\ \widehat{\sigma}_1) &= \mathrm{gev}(\widehat{\sigma}_1)\\ \mathrm{gev}(\forall \beta.\ \widehat{\sigma}_1) &= \{\beta\} \cup \mathrm{gev}(\widehat{\sigma}_1).\end{aligned}$$
3. The set of *qualified effect variables* of a type scheme $\widehat{\sigma}$, written $\mathrm{qev}(\widehat{\sigma})$, is given by
$$\begin{aligned}\mathrm{qev}(\widehat{\tau}) &= \emptyset\\ \mathrm{qev}(\pi \Rightarrow \widehat{\rho}_1) &= \mathrm{fev}(\pi) \cup \mathrm{qev}(\widehat{\rho}_1)\\ \mathrm{qev}(\forall \alpha.\ \widehat{\sigma}_1) &= \mathrm{qev}(\widehat{\sigma}_1)\\ \mathrm{qev}(\forall \beta.\ \widehat{\sigma}_1) &= \mathrm{qev}(\widehat{\sigma}_1).\end{aligned}$$ ∎

Definition 10. A type scheme $\widehat{\sigma}$ is *unambiguous*, if $\mathrm{gev}(\widehat{\sigma}) \cap \mathrm{qev}(\widehat{\sigma}) \subseteq \mathrm{aev}(\widehat{\sigma})$. ∎

The "best" completion of a source term is now given by a derivation of its most general unambiguous, satisfiable type scheme.

Definition 11. An unambiguous, satisfiable solution $\widehat{\sigma}$ is a *principal unambiguous, satisfiable type scheme* of $\widehat{t}$ in $\widehat{\Gamma}$, if $\widehat{\sigma}$ is more general than every unambiguous, satisfiable solution of $\widehat{t}$ in $\widehat{\Gamma}$. ∎

Theorem 12 (Principal Type Schemes). If a term $\widehat{t}$ has an unambiguous, satisfiable solution in $\widehat{\Gamma}$, then it has a principal unambiguous, satisfiable type scheme in $\widehat{\Gamma}$. ∎

Disambiguation is crucial for the effectiveness of our sharing analysis: without it, no closure will ever be marked as nonupdatable. Furthermore, aggressive disambiguation has a positive impact on the performance of inference algorithms, since it allows for smaller predicate sets. Predicate sets are also kept small because we restrict our attention to satisfiable type schemes, because assuming satisfiability enables us to subject predicates to *improvement* (Jones 1995): if we encounter predicates of the form $\beta \sqsubseteq 1$ or $\omega \sqsubseteq \beta$, we infer that β is to be instantiated to, respectively, 1 or ω. Furthermore, a related technique, *simplification*, can be applied to safely remove trivial predicates like $1 \sqsubseteq \beta$, $\beta \sqsubseteq \beta$, and $\beta \sqsubseteq 1$ as well as predicates that are already implied by the transitivity of the ordering on effects.

The incremental type-reconstruction algorithm of Jones (1995), itself an extension of Algorithm W (Damas and Milner 1982), can now be adapted for inferring appropriate principal type schemes for terms and simultaneously reconstructing the associated completions.

5.4 Applicability

The type and effect system of Figure 6 is a conservative extension of the Hindley-Milner system. Moreover, every term that is typeable in the Hindley-Milner system is also typeable in the type and effect system.

Writing $\lfloor \widehat{\sigma} \rfloor_{\mathrm{HM}}$ and $\lfloor \widehat{\Gamma} \rfloor_{\mathrm{HM}}$ for the type scheme and context obtained by stripping all annotations and predicates and such from, respectively, $\widehat{\sigma}$ and $\widehat{\Gamma}$, and $\Gamma^{\mathrm{HM}} \vdash_{\mathrm{HM}} \widehat{t} : \sigma^{\mathrm{HM}}$ for typing judgements in the Hindley-Milner system, we have:

Theorem 13 (Conservative Extension).

1. If $\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \widehat{\sigma}$, then $\lfloor \widehat{\Gamma} \rfloor_{\mathrm{HM}} \vdash_{\mathrm{HM}} \widehat{t} : \lfloor \widehat{\sigma} \rfloor_{\mathrm{HM}}$.
2. If $\Gamma^{\mathrm{HM}} \vdash_{\mathrm{HM}} \widehat{t} : \sigma^{\mathrm{HM}}$, then there exist $\widehat{\Gamma}$, $\widehat{\sigma}$, and φ, such that $\lfloor \widehat{\Gamma} \rfloor_{\mathrm{HM}} = \Gamma^{\mathrm{HM}}$, $\lfloor \widehat{\sigma} \rfloor = \sigma^{\mathrm{HM}}$, and $\widehat{\Gamma} \vdash \widehat{t} :^{\varphi} \widehat{\sigma}$ with $\widehat{\Gamma}$ consistent, and $\widehat{\sigma}$ unambiguous and satisfiable in $\widehat{\Gamma}$. ∎

6. Related Work

Early work on sharing analysis is due to Goldberg (1987) who uses abstract interpretation to derive information about the usage of partial applications. Marlow (1994) also defines an abstract interpretation; he measures the number of updates performed by a naive implementation of call-by-need evaluation and estimates that, for a typical program, 70% of all such updates are actually unnecessary.

Launchbury et al. (1992) formulate a sharing analysis in terms of a substructural type system. Their analysis distinguishes between terms that are used never, terms that are used at most once, and

terms that may be used more than once. Turner et al. (1995) present a comparable type-based analysis, that is less conservative but does not keep track of terms that are never used. Mogensen (1998) presents an adaptation that does detect unused terms.

Another extension to the system of Turner et al. is given by Wansbrough and Peyton Jones (1999). In contrast to the aforementioned analyses, their analysis operates on a polymorphically typed source language with user-defined algebraic data types. Moreover, their type system incorporates a subtyping relation, which makes that the analysis is, to some extent, context sensitive. In practice, however, the analysis turns out to perform rather poorly, especially in the context of curried function definitions. The proposed solution to this problem is to extend the type system with a limited form of polyvariance that is dubbed *simple polymorphism* (Wansbrough and Peyton Jones 2000; Wansbrough 2002). Simple polymorphism restricts the quantified effect variables of a type scheme to those that simultaneously occur in covariant and contravariant positions. Furthermore, it does not permit quantified type variables to be subjected to subtype coercions. The main motivation for the inclusion of simple polymorphism is that it adds of the power of full polyvariance, while it allows for a combination with subtyping that does not suffer from the complications that arise when full polyvariance is to be mixed with subtyping. Still, an inference algorithm for simple polymorphism is complicated in its own right.

Our approach to keeping the mixture of polyvariance and subsumption manageable is, in a sense, opposite to that of Wansbrough and Peyton Jones. Instead of employing a more limited form of polyvariance, we employ a more limited form of subsumption, i.e., we do not extend the subeffecting relation to a full subtyping relation. While an analysis based on simple polymorphism behaves, for a large class of realistic programs, more conservatively than systems that combine full polyvariance and subtyping (e.g., Gustavsson and Svenningsson 2001; Gustavsson 2001), the combination of full polyvariance and subeffecting is, in practice, just as expressive as full polyvariance and shape-conformant subtyping.

Gedell et al. (1999) measure the impact of polyvariance, subtyping, whole program analysis, and the treatment of user-defined algebraic data types on the performance of type-based usage analyses. They show that all these features increase the precision of the analysis, but that acceptable results can still be obtained if one of them is left out. They do not address the option of replacing full subtyping by mere subeffecting. Our use of qualified types allows for an implementation by means of an incremental inference algorithm. Hence, our analysis can, almost effortlessly, be used in a setting that demands separate compilation; it then conservatively assumes that top-level definitions may be used more than once. Still, analysis results will be better if they are obtained by means of a whole program analysis. We have not yet addressed the analysis of user-defined algebraic data types: although the details remain future work, we expect to encounter the same trade-offs as described by Gedell et al.

Uniqueness typing was first described for a monovariant system by Barendsen and Smetsers (1993). Later, the same authors added uniqueness polymorphism, i.e., polyvariance (Barendsen and Smetsers 1995). Recently, de Vries et al. (2007) defined a uniqueness type system that incorporates rank-n polymorphism.

Most of the existing systems for sharing analysis and uniqueness typing come with real-life implementations and experimental data. Our system has not reached this level of maturity yet; implementing our analysis in a large-scale compiler is left as future work.

The similarity between type-based sharing analyses and systems with uniqueness typing was first signalled by Wansbrough and Peyton Jones (1999, Section 2.2), who observed a degree of duality between the two analyses and stated that "it would be interesting to see whether the duality can be made more precise". We believe to have done so in this paper.

7. Conclusions and Further Work

We have formulated a generic usage analysis with subeffecting and polyvariance, that can be instantiated to both sharing analysis and uniqueness typing. Building the analysis upon the theory of qualified types enables us to reuse a great deal of metatheoretical tools and implementation techniques.

The Clean language, for example, features both type classes and uniqueness typing. With our approach, a Clean compiler can implement both type classes and uniqueness typing on top of a common infrastructure for qualified types. Furthermore, the error messages that the existing implementation of Clean produces for programs with incorrect uniqueness properties are often not very descriptive. Heeren (2005) describes a type-inference framework that aims at producing understandable type-error messages. The part of his framework that deals with type inferencing in the context of type classes applies, essentially, to all systems with qualifed types. We are therefore planning to implement our analysis within Heeren's framework; we hope to obtain a system for uniqueness typing that produces type-error messages that are better suited to the needs of the programmer than those produced by the Clean compiler.

Another interesting direction for future work is to extend the effect language of our system with a constant that expresses that a value is never used. This additional expressiveness has already been put to use for sharing analysis (Launchbury et al. 1992; Mogensen 1998), but has not yet been explored in the context of uniqueness typing.

Finally, we plan to investigate to what extent our approach to context-sensitive usage analyses is applicable to other program analyses, such as binding-time analysis and strictness analysis.

Acknowledgments

The authors would like to thank Edsko de Vries and Rinus Plasmeijer for discussing with us the tension between subsumption and containment for function types. We are also grateful to the four anonymous referees for their detailed and constructive comments. This work was supported in part by the Netherlands Organization for Scientific Research through its project on "Scriptable Compilers" (612.063.406) and by Microsoft Research through its European PhD Scholarship Programme.

References

Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems. In R. K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science, 13th Conference, Bombay, India, December 15–17, 1993, Proceedings*, volume 761 of *Lecture Notes in Computer Science*, pages 41–51. Springer-Verlag, 1993.

Erik Barendsen and Sjaak Smetsers. Uniqueness type inference. In Manuel V. Hermenegildo and S. Doaitse Swierstra, editors, *Programming Languages: Implementation, Logics and Programs, 7th International Symposium, PLILP'95, Utrecht, The Netherlands, September 20–22, 1995, Proceedings*, volume 982 of *Lecture Notes in Computer Science*, pages 189–206. Springer-Verlag, 1995.

Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1982*, pages 207–212. ACM Press, 1982.

Edsko de Vries, Rinus Plasmeijer, and David Abrahamson. Uniqueness typing redefined. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *Implementation and Application of Functional Languages,*

18th International Workshop, IFL 2006, Budapest, Hungary, September 4–6, 2006, Revised Selected Papers, volume 4449 of *Lecture Notes in Computer Science*, pages 181–198. Springer-Verlag, 2007.

Tobias Gedell, Jörgen Gustavsson, and Josef Svenningsson. Polymorphism, subtyping, whole program analysis and accurate data types in usage analysis. In Naoki Kobayashi, editor, *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8–10, 2006, Proceedings*, volume 4279 of *Lecture Notes in Computer Science*, pages 200–216. Springer-Verlag, 1999.

John M. Gifford, David K. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, August 4–6, 1986, Cambridge, Massachusetts, USA*, pages 28–38. ACM Press, 1986.

Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

Benjamin Goldberg. Detecting sharing of partial applications in functional programs. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14–16, 1987, Proceedings*, volume 274 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1987.

Jörgen Gustavsson. *Space-Safe Transformation and Usage Analysis for Call-by-Need Languages*. PhD thesis, Göteborg University, 2001.

Jörgen Gustavsson. A type based sharing analysis for update avoidance and optimisation. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27–29, 1998*, pages 39–50. ACM Press, 1999.

Jörgen Gustavsson and David Sands. A foundation for space-safe transformations of call-by-need programs. *Electronic Notes in Theoretical Computer Science*, 26:69–86, 1999.

Jörgen Gustavsson and Josef Svenningsson. A usage analysis with bounded usage polymorphism and subtyping. In Markus Mohnen and Pieter W. M. Koopman, editors, *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Aachen, Germany, September 4–7, 2000, Selected Papers*, volume 2011 of *Lecture Notes in Computer Science*, pages 140–157. Springer-Verlag, 2001.

Bastiaan Heeren. *Top Quality Type Error Messages*. PhD thesis, Utrecht University, 2005.

J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the Americal Mathematical Society*, 146:29–60, 1969.

Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, 1994.

Mark P. Jones. Simplifying and improving qualified types. In *Conference Record of FPCA'95 SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture. La Jolla, CA, USA, 25–28 June 1995*, pages 160–169. ACM Press, 1995.

Stefan Kaes. Parametric overloading in polymorphic programming languages. In Harald Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21–24, 1988, Proceedings*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144. Springer-Verlag, 1988.

Naoki Kobayashi. Quasi-linear types. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 20–22, 1999, San Antonio, TX*, pages 29–42. ACM Press, 1999.

John Launchbury, Andy Gill, John Hughes, Simon Marlow, Simon Peyton Jones, and Philip Wadler. Avoiding unnecessary updates. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6–8 July 1992*, Workshops in Computing, pages 144–153. Springer-Verlag, 1992.

Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark Shields. Implicit parameters: Dynamic scoping with static types. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 19–21, 2000, Boston, Massachusetts*, pages 108–118. ACM Press, 2000.

Simon Marlow. Update avoidance analysis by abstract interpretation. In Kevin Hammond and John T. O'Donnell, editors, *1993 Glasgow Workshop on Functional Programming, Ayr*, pages 170–184. Springer-Verlag, 1994.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

Torben Æ. Mogensen. Types for 0, 1 or many uses. In Chris Clack, Kevin Hammond, and Anthony J. T. Davie, editors, *Implementation of Functional Languages, 9th International Workshop, IFL'97, St. Andrews, Scotland, UK, September 10–12, 1997, Selected Papers*, volume 1467 of *Lectures Notes in Computer Science*, pages 112–122. Springer-Verlag, 1998.

Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Enst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer-Verlag, 1999.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, 2003.

Rinus Plasmeijer and Marco van Eekelen. Concurrent Clean language report—version 1.3. Technical Report CSI-R9816, University of Nijmegen, 1998.

John C. Reynolds. Towards a theory of type structure. In Bernard Robinet, editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9–11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.

Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

Jean-Pierre Talpin and Pierre Jouvelot. A type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Conference Record of FPCA '95 SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture. La Jolla, CA, USA, 25–28 June 1995*, pages 1–11. ACM Press, 1995.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, January 1989*, pages 60–76. ACM Press, 1989.

Keith Wansbrough. *Simple Polymorphic Usage Analysis*. PhD thesis, University of Cambridge, 2002.

Keith Wansbrough and Simon Peyton Jones. Simple usage polymorphism. In *Preliminary Proceedings of the Third Workshop on Types in Compilation (TIC 2000)*, 2000. The proceedings of the workshop have been published as a technical report (CMU-CS-00-161) at Carnegie Mellon University.

Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 20–22, 1999, San Antonio, TX*, pages 15–18. ACM Press, 1999.