Type and Effect Systems

Flemming Nielson & Hanne Riis Nielson

Department of Computer Science, Aarhus University, Denmark.

Abstract. The design and implementation of a correct system can benefit from employing static techniques for ensuring that the dynamic behaviour satisfies the specification. Many programming languages incorporate types for ensuring that certain operations are only applied to data of the appropriate form. A natural extension of type checking techniques is to enrich the types with annotations and effects that further describe intensional aspects of the dynamic behaviour.

Keywords. Polymorphic type systems, effect annotations, subeffecting and subtyping, semantic correctness, type inference algorithms, syntactic soundness and completeness. Analyses for control flow, binding times, side effects, region structure, and communication structure.

1 Introduction

Static analysis of programs comprises a broad collection of techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically during computation; this may be used to validate program transformations, to generate more efficient code or to increase the understanding of the software so as to demonstrate that the software satisfies its specification. We shall find it helpful to divide the techniques into the following two approaches. The *flow based* approach includes the traditional data flow analysis techniques for mainly imperative and object-oriented languages; it also includes the constraint based control flow analysis techniques developed for functional and object oriented languages; finally, it includes the use of mathematical modelling in the abstract interpretation of imperative, functional, concurrent and logic languages. The *inference based* approach includes general logical techniques touching upon program verification and model checking; it also includes type and effect systems developed for functional, imperative and concurrent languages and it is this latter group of techniques that we consider here.

We shall suppose that a typed programming language is given. In *soft typing* all programs can be typed because a "top" type can be used in the absence of meaningful "ordinary" types; this perspective is similar to that of the flow based approach and is quite useful for analysing the behaviour of programs but is less useful for enforcing the absence of dynamic errors. In this paper we focus on *strong typing* where no "top" type is around and where certain erroneous

programs are rejected by the type system; in this way types are not only used for analysing the behaviours of programs but also for enforcing the absence of certain kinds of dynamic errors.

The overall approach of type systems is to associate types to programs; normally the types merely describe the form of the data supplied to, and produced by, programs. The association of types to programs is done by a set of inference rules that are largely syntax-directed; since subprograms may contain free variables this needs to be relative to a type environment that maps the free variables to their types. We express the typing by means of a typing judgement that is usually a ternary relation on type environments, programs and types; it is frequently written $\Gamma \vdash p : \tau$ where p is the program, τ is the type, and Γ is the type environment. The main challenges of devising type systems is *(i)* to ensure that they are semantically correct (with respect to some dynamic semantics), *(ii)* to ensure that they are decidable so that types can be checked by an algorithm, and *(iii)* to ensure that there always is a "best" type, called a principal type, so that an algorithm can produce the intended type automatically.

Type and effect systems refine the type information by annotating the types so as to express further intensional or extensional properties of the semantics of the program [18–21]. In Section 2 this takes the form of annotating the base types or the type constructors. In Section 3 we study effect systems where the annotations describe certain intensional aspects of the actions taking place during evaluation. In Section 4 we further enrich the expressivenes of effects so as to obtain causal information in the manner of process algebras. We then expose the overall methodology behind type and effect systems in Section 5 and indicate those combinations of features that challenge state-of-the-art.

Further Reading. A more thorough development of the techniques of static analysis can be found in [30] (particularly in Chapter 5 that deals with type and effect systems) as well as in the references given.

2 Annotated Type Systems

Many programming languages incorporate types as a static technique for ensuring that certain operations are only applied to data of the appropriate form; this is useful for ensuring that the dynamic behaviour satisfies the specification.

Example 1. A Typed Language.

We shall use the following simple functional language to illustrate the development; constructs for iteration, recursion and conditionals present no obstacles and have only been left out in the interest of brevity. We shall later extend the language with side effects (as in Standard ML) and communication (as in Concurrent ML) thereby suggesting that type and effect systems apply equally well to imperative and concurrent languages.

The language has expressions (or programs) e and types τ given by:

$$e ::= c \mid x \mid \mathtt{fn}_\pi\ x\ \mathtt{=>}\ e_0 \mid e_1\ e_2 \mid \cdots$$
$$\tau ::= \mathtt{int} \mid \mathtt{bool} \mid \cdots \mid \tau_1 \to \tau_2$$

Here c denotes a family of constants (each of type τ_c), x denotes a family of variables, and π is an identification of the function abstraction to be used in the control flow analysis to be presented in Example 2.

The typing judgements of the underlying or original type system have the form $\Gamma \vdash_{\mathsf{UL}} e : \tau$ where the type environment Γ maps variables to types; the definition is as follows:

$$\Gamma \vdash_{\mathsf{UL}} c : \tau_c \qquad \frac{\Gamma[x \mapsto \tau_x] \vdash_{\mathsf{UL}} e_0 : \tau_0}{\Gamma \vdash_{\mathsf{UL}} \mathtt{fn}_\pi\ x\ \mathtt{=>}\ e_0 : \tau_x \to \tau_0}$$

$$\Gamma \vdash_{\mathsf{UL}} x : \Gamma(x) \qquad \frac{\Gamma \vdash_{\mathsf{UL}} e_1 : \tau_2 \to \tau_0 \quad \Gamma \vdash_{\mathsf{UL}} e_2 : \tau_2}{\Gamma \vdash_{\mathsf{UL}} e_1\ e_2 : \tau_0}$$

That a function has type $\tau_1 \to \tau_2$ means that given an argument of type τ_1 it will return a value of type τ_2 in case it terminates. □

Perhaps the simplest technique for extending the expressiveness of types is to add annotations to the type constructors or base types. One popular class of analyses that can be expressed using this technique consists of interprocedural control flow analyses which track the origins of where functions might have been defined [7, 8, 11]; this can be extended with components for tracking where functions are applied and thus has strong similarities to the classical *use-definition* and *definition-use* analyses of data flow analysis.

Example 2. Control Flow Analysis.

To obtain a control flow analysis we shall annotate the function type $\tau_1 \to \tau_2$ with information, φ, about which function it might be:

$$\varphi ::= \{\pi\} \mid \varphi_1 \cup \varphi_2 \mid \emptyset$$
$$\widehat{\tau} ::= \mathtt{int} \mid \mathtt{bool} \mid \cdots \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2$$

So φ will be a *set* of function names – describing the set of function definitions that can result in a function of a given type.

The typing judgements of the control flow analysis have the form $\widehat{\Gamma} \vdash_{\mathsf{CFA}} e : \widehat{\tau}$ where the type environment $\widehat{\Gamma}$ maps variables to annotated types. The judgements are defined in Table 1; note that the clause for function abstraction annotates the arrow of the resulting function type with the information that the abstraction named π should be included in the set $\{\pi\} \cup \varphi$ of functions that could be returned. In the presence of conditionals it is essential that we use $\{\pi\} \cup \varphi$

$$\widehat{\Gamma} \vdash_{\mathsf{CFA}} c : \widehat{\tau}_c \qquad \frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_x] \vdash_{\mathsf{CFA}} e_0 : \widehat{\tau}_0}{\widehat{\Gamma} \vdash_{\mathsf{CFA}} \mathtt{fn}_\pi\ x\ \mathtt{=>}\ e_0 : \widehat{\tau}_x \xrightarrow{\{\pi\} \cup \varphi} \tau_0}$$

$$\widehat{\Gamma} \vdash_{\mathsf{CFA}} x : \widehat{\Gamma}(x) \qquad \frac{\widehat{\Gamma} \vdash_{\mathsf{CFA}} e_1 : \widehat{\tau}_2 \xrightarrow{\varphi} \widehat{\tau}_0 \quad \widehat{\Gamma} \vdash_{\mathsf{CFA}} e_2 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\mathsf{CFA}} e_1\ e_2 : \widehat{\tau}_0}$$

Table 1. Control Flow Analysis: $\widehat{\Gamma} \vdash_{\mathsf{CFA}} e : \widehat{\tau}$ (Example 2).

rather than $\{\pi\}$ because the latter choice does not give rise to a *conservative extension* of the underlying type system: this means that there will be expressions that are typed in the underlying type system but that have no analysis in the control flow analysis; (this point is related to the issue of subeffecting to be discussed in Section 3.)

We should point out that we allow to replace $\widehat{\tau}_1 \xrightarrow{\varphi_1} \widehat{\tau}_2$ by $\widehat{\tau}_1 \xrightarrow{\varphi_2} \widehat{\tau}_2$ whenever φ_1 and φ_2 are "equal as sets". More generally we allow to replace $\widehat{\tau}_1$ by $\widehat{\tau}_2$ if they have the same underlying types and all annotations on corresponding function arrows are "equal as sets". To be utterly formal this can be axiomatised by a set of axioms and rules expressing that set union has a unit and is idempotent, commutative, and associative, together with axioms and rules ensuring that equality is an equivalence relation as well as a congruence; the abbreviation *UCAI* is often used for these properties. □

Subtyping and Polymorphism

Another popular class of analyses that can be expressed by annotations is the binding time analyses (e.g. [13]) which distinguish data as to whether they are static (available at compile-time) or dynamic (available at run-time); these analyses form the basis of *partial evaluation* and can also be used as the basis for *security* analyses (e.g. [12]) that distinguish between secret and public information.

Example 3. Binding Time Analysis.

For binding time analysis we extend the language of Examples 1 and 2 with a `let`-construct:

$$e ::= \cdots \mid \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2$$

(In fact $\mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2$ is semantically equivalent to $(\mathtt{fn}\ x\ \mathtt{=>}\ e_2)\ e_1$.) The annotations of interest for the binding time analysis are:

$$\begin{aligned}
\varphi &::= \beta \mid \mathsf{S} \mid \mathsf{D} \\
\widehat{\tau} &::= \mathtt{int}^{\varphi} \mid \mathtt{bool}^{\varphi} \mid \cdots \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \\
\widehat{\sigma} &::= \forall(\beta_1, \cdots, \beta_n).\widehat{\tau} \mid \widehat{\tau}
\end{aligned}$$

The annotation S is used to model data that is available statically, D is used to model data that is available dynamically, and β is an annotation variable that can take its values among S and D.

A partial ordering on annotations $\varphi \sqsubseteq \varphi'$ may be defined by:

$$\varphi \sqsubseteq \varphi \qquad \mathsf{S} \sqsubseteq \mathsf{D}$$

Types contain annotations on the type constructors as well as on the base types; a static function operating on dynamic integers will thus have the annotated type $\mathtt{int}^{\mathsf{D}} \xrightarrow{\mathsf{S}} \mathtt{int}^{\mathsf{D}}$. This type system is motivated by applications to partial evaluation and this suggests imposing a well-formedness condition on types so as to rule out types like $\mathtt{int}^{\mathsf{S}} \xrightarrow{\mathsf{D}} \mathtt{int}^{\mathsf{S}}$ that are regarded as being meaningless. This is performed by the auxiliary judgement $\widehat{\tau} \triangleright \varphi$ that additionally extracts the top-level annotation φ from the annotated type $\widehat{\tau}$:

$$\mathtt{int}^{\varphi} \triangleright \varphi \qquad \mathtt{bool}^{\varphi} \triangleright \varphi$$

$$\frac{\widehat{\tau}_1 \triangleright \varphi_1 \quad \widehat{\tau}_2 \triangleright \varphi_2}{\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \triangleright \varphi} \text{ if } \varphi \sqsubseteq \varphi_1 \text{ and } \varphi \sqsubseteq \varphi_2$$

In short, a static function is allowed to operate on dynamic data but not vice versa. Since we have annotation variables we can express a limited form of annotation polymorphism and we use $\widehat{\sigma}$ to denote the corresponding type schemes; for simplicity we do not incorporate type variables or type polymorphism.

The typing judgements have the form $\widehat{\Gamma} \vdash_{\mathsf{BTA}} e : \widehat{\sigma}$ where the type environment $\widehat{\Gamma}$ maps variables to type schemes (or types) and $\widehat{\sigma}$ is the type scheme (or type) for the expression e. The analysis is specified by the axioms and rules of Table 2 and is explained in the sequel. The first five axioms and rules are straightforward; note that the rule for function abstraction checks that the type is well-formed and that the rule for `let` makes use of type schemes.

The next rule is a subtyping rule that allows to weaken the information contained in an annotated type. The subtype ordering $\widehat{\tau} \leq \widehat{\tau}'$ is given by:

$$\mathtt{int}^{\varphi} \leq \mathtt{int}^{\varphi'} \text{ if } \varphi \sqsubseteq \varphi'$$

$$\mathtt{bool}^{\varphi} \leq \mathtt{bool}^{\varphi'} \text{ if } \varphi \sqsubseteq \varphi'$$

$$\frac{\widehat{\tau}_1' \leq \widehat{\tau}_1 \quad \widehat{\tau}_2 \leq \widehat{\tau}_2'}{\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \leq \widehat{\tau}_1' \xrightarrow{\varphi'} \widehat{\tau}_2'} \text{ if } \varphi \sqsubseteq \varphi' \ \wedge \ \widehat{\tau}_1' \xrightarrow{\varphi'} \widehat{\tau}_2' \triangleright \varphi'$$

This ensures that only well-formed types are produced and that the ordering is reversed for arguments to functions; we say that $\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2$ is *contravariant* in $\widehat{\tau}_1$ but *covariant* in φ and $\widehat{\tau}_2$. (Think of the annotated type $\widehat{\tau} \xrightarrow{\varphi'} \widehat{\tau}'$ as being analogous to the logical formula $\widehat{\tau} \Rightarrow \varphi' \wedge \widehat{\tau}'$ and use that the inference rule expresses the monotonicity of logical implication.)

The final two rules are responsible for the polymorphism. The first rule is the *generalisation rule* that is used to construct type schemes: we can quantify over any

$$\widehat{\Gamma} \vdash_{\mathsf{BTA}} c : \widehat{\tau}_c \qquad \widehat{\Gamma} \vdash_{\mathsf{BTA}} x : \widehat{\Gamma}(x)$$

$$\frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_x] \vdash_{\mathsf{BTA}} e_0 : \widehat{\tau}_0}{\widehat{\Gamma} \vdash_{\mathsf{BTA}} \mathtt{fn}_\pi\ x\ \mathtt{=>}\ e_0 : \widehat{\tau}_x \xrightarrow{\varphi} \widehat{\tau}_0} \quad \text{if } (\widehat{\tau}_x \xrightarrow{\varphi} \widehat{\tau}_0) \triangleright \varphi$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{BTA}} e_1 : \widehat{\tau}_2 \xrightarrow{\varphi} \widehat{\tau}_0 \qquad \widehat{\Gamma} \vdash_{\mathsf{BTA}} e_2 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\mathsf{BTA}} e_1\ e_2 : \widehat{\tau}_0}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{BTA}} e_1 : \widehat{\sigma}_1 \quad \widehat{\Gamma}[x \mapsto \widehat{\sigma}_1] \vdash_{\mathsf{BTA}} e_2 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\mathsf{BTA}} \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 : \widehat{\tau}_2}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{BTA}} e : \widehat{\tau}}{\widehat{\Gamma} \vdash_{\mathsf{BTA}} e : \widehat{\tau}'} \quad \text{if } \widehat{\tau} \leq \widehat{\tau}'$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{BTA}} e : \widehat{\tau}}{\widehat{\Gamma} \vdash_{\mathsf{BTA}} e : \forall(\beta_1, \cdots, \beta_n).\widehat{\tau}} \quad \text{if } \beta_1, \cdots, \beta_n \text{ do not occur free in } \widehat{\Gamma}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{BTA}} e : \forall(\beta_1, \cdots, \beta_n).\widehat{\tau}}{\widehat{\Gamma} \vdash_{\mathsf{BTA}} e : (\theta\ \widehat{\tau})} \quad \text{if } \mathit{dom}(\theta) \subseteq \{\beta_1, \cdots, \beta_n\} \text{ and } \exists\varphi : (\theta\ \widehat{\tau}) \triangleright \varphi$$

Table 2. Binding Time Analysis: $\widehat{\Gamma} \vdash_{\mathsf{BTA}} e : \widehat{\tau}\ \&\ \varphi$ (Example 3).

annotation variable that does not occur free in the type environment; this rule is usually used immediately before the rule for the `let`-construct. The second rule is the *instantiation rule* that can be used to turn type schemes into annotated types: we just apply a substitution in order to replace the bound annotation variables with other annotations; this rule is usually used immediately after the axiom for variables. □

References for type systems with subtyping include [9, 10, 23] as well as the more advanced [16, 37, 38] that also deal with Hindley/Milner polymorphism (as found in Standard ML). To allow a general treatment of subtyping, these papers generally demand constraints to be an explicit part of the inference system and this is somewhat more complex than the approach taken here; such considerations are mainly motivated by the desire to obtain principal types and in order to develop syntactically sound and complete type inference algorithms as will be discussed in Section 5. Indeed, our formulation of subtyping only allows *shape conformant subtyping*, where the underlying type system does *not* make use of any form of subtyping, and is thus somewhat simpler than *atomic subtyping*, where an ordering is imposed upon base types, and *general subtyping*, where an ordering may be imposed between arbitrary types.

Strictness analyses and classical data flow analyses can also be expressed as annotated type systems but to be useful they may require the type system to be extended with conjunction or disjunction types [4, 5, 14, 15] thereby touching

upon the logical techniques. In annotated type systems, as well as in the type and effect systems considered below, the annotations are normally sets of some kind, but linking up with abstract interpretation it should be possible to allow more general annotations that are elements of a complete lattice (that is possibly of finite height as in the "monotone frameworks" of data flow analysis); however, this possibility is hardly considered in the literature except in the case of binding time analysis where the binding times (e.g. static and dynamic) are partially ordered, c.f. [13, 24].

3 Type and Effect Systems

The typing judgements of type systems take the following general form: a type is associated with a program (or an expression or a statement) relative to a type environment providing the type (or type scheme) for each free variable; this also holds for the typing judgements used for the annotated type systems presented above. Effect systems can be viewed as an outgrowth of annotated type system where the typing judgements take the following more elaborate form: a type *and an effect* is associated with a program relative to a type environment. Formally, effects are nothing but the annotations already considered, but conceptually, they describe intensional information about what takes place during evaluation of the program unlike what was the case above.

Subeffecting and Subtyping

The literature has seen a great variation in the uses to which effects have been put: collecting the set of procedures or functions called [41], collecting the set of storage cells written or read during execution [40], determining what exceptions can be raised during evaluation, and collecting the regions in which evaluation takes place [44] to mention just a few. We begin by considering an analysis for collecting the set of storage cells written or read during execution.

Example 4. Adding Imperative Constructs.

To facilitate the side effect analysis we shall add imperative constructs (resembling those of Standard ML) for creating reference variables and for accessing and updating their values:

$$e ::= \cdots \mid \texttt{new}_\pi\ x\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{!}x \mid x\ \texttt{:=}\ e_0$$

The idea is that $\texttt{new}_\pi\ x\texttt{:=}\ e_1\ \texttt{in}\ e_2$ creates a new reference variable x for use in e_2 and initialises it to the value of e_1; as above we use π to identify the creation point. The value of the reference variable x can be obtained by writing $\texttt{!}x$ and it may be set to a new value by the assignment $x\ \texttt{:=}\ e_0$. The type of a reference

cell for values of type τ is τ `ref` and the underlying type system of Example 1 is extended with the rules:

$$\Gamma \vdash_{\mathsf{UL}} \texttt{!}x : \tau \text{ if } \Gamma(x) = \tau \texttt{ ref}$$

$$\frac{\Gamma \vdash_{\mathsf{UL}} e : \tau}{\Gamma \vdash_{\mathsf{UL}} x \texttt{ := } e : \tau} \text{ if } \Gamma(x) = \tau \texttt{ ref}$$

$$\frac{\Gamma \vdash_{\mathsf{UL}} e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1 \texttt{ ref}] \vdash_{\mathsf{UL}} e_2 : \tau_2}{\Gamma \vdash_{\mathsf{UL}} \texttt{new}_\pi\ x \texttt{ := } e_1 \texttt{ in } e_2 : \tau_2}$$

Example 5. Side Effect Analysis.

In the side effect analysis a reference variable is represented by a set ϱ of program points where it could have been created; this set is called a region and has the general form $\{\pi_1\} \cup \cdots \cup \{\pi_n\}$ which we write as the set $\{\pi_1, \cdots, \pi_n\}$. The annotations of interest are:

$$\begin{aligned}
\varphi &::= \{\texttt{!}\pi\} \mid \{\pi\texttt{:=}\} \mid \{\texttt{new}\,\pi\} \mid \varphi_1 \cup \varphi_2 \mid \emptyset \\
\varrho &::= \{\pi\} \mid \varrho_1 \cup \varrho_2 \mid \emptyset \\
\widehat{\tau} &::= \texttt{int} \mid \texttt{bool} \mid \cdots \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \mid \widehat{\tau}\ \texttt{ref}\ \varrho
\end{aligned}$$

Here $\widehat{\tau}$ `ref` ϱ is the type of a location created at one of the program points in the region ϱ; the location is used for holding values of the annotated type $\widehat{\tau}$. The annotation $\texttt{!}\pi$ means that the value of a location created at π is accessed, $\pi\texttt{:=}$ means that a location created at π is assigned, and $\texttt{new}\,\pi$ that a new location has been created at π.

The typing judgements have the form $\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau}\ \&\ \varphi$. This means that under the type environment $\widehat{\Gamma}$, if the expression e terminates then the resulting value will have the annotated type $\widehat{\tau}$ and φ describes the side effects that might have taken place during evaluation. As before the type environment $\widehat{\Gamma}$ will map variables to annotated types; no effects are involved because the semantics is *eager* rather than *lazy*.

The analysis is specified by the axioms and rules of Table 3; these rules embody the essence of effect systems. In the clauses for constants and variables we record that there are no side effects so we use $\emptyset$ for the overall effect. The premise of the clause for function abstraction gives the effect of the function body and this effect is used to annotate the arrow of the function type whereas we use $\emptyset$ as the overall effect of the function definition itself: no side effects can be observed by simply defining the function. In the rule for function application we see how the information comes together: the overall effect is what can be observed from evaluating the argument e_1, what can be observed from evaluating the argument e_2, and what is obtained from evaluating the body of the function called.

Turning to the rules involving reference variables we make sure that we only assign a value of the appropriate type to the reference variable. Also, in each of

$$\widehat{\Gamma} \vdash_{\mathsf{SE}} c : \widehat{\tau}_c \;\&\; \emptyset \qquad\qquad \widehat{\Gamma} \vdash_{\mathsf{SE}} x : \widehat{\Gamma}(x) \;\&\; \emptyset$$

$$\frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_x] \vdash_{\mathsf{SE}} e_0 : \widehat{\tau}_0 \;\&\; \varphi_0}{\widehat{\Gamma} \vdash_{\mathsf{SE}} \mathtt{fn}_\pi\ x\ \mathtt{=>}\ e_0 : \widehat{\tau}_x \xrightarrow{\varphi_0} \widehat{\tau}_0 \;\&\; \emptyset}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e_1 : \widehat{\tau}_2 \xrightarrow{\varphi_0} \widehat{\tau}_0 \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{SE}} e_2 : \widehat{\tau}_2 \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{SE}} e_1\ e_2 : \widehat{\tau}_0 \;\&\; \varphi_1 \cup \varphi_2 \cup \varphi_0}$$

$$\widehat{\Gamma} \vdash_{\mathsf{SE}} \mathtt{!}x : \widehat{\tau} \;\&\; \{\mathtt{!}\pi_1, \cdots, \mathtt{!}\pi_n\} \text{ if } \widehat{\Gamma}(x) = \widehat{\tau}\ \mathtt{ref}\ \{\pi_1, \cdots, \pi_n\}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau} \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{SE}} x\ \mathtt{:=}\ e : \widehat{\tau} \;\&\; \varphi \cup \{\pi_1\mathtt{:=}, \cdots, \pi_n\mathtt{:=}\}} \text{ if } \widehat{\Gamma}(x) = \widehat{\tau}\ \mathtt{ref}\ \{\pi_1, \cdots, \pi_n\}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e_1 : \widehat{\tau}_1 \;\&\; \varphi_1 \quad \widehat{\Gamma}[x \mapsto\ \widehat{\tau}_1\ \mathtt{ref}(\varrho \cup \{\pi\})] \vdash_{\mathsf{SE}} e_2 : \widehat{\tau}_2 \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{SE}} \mathtt{new}_\pi\ x\ \mathtt{:=}\ e_1\ \mathtt{in}\ e_2 : \widehat{\tau}_2 \;\&\; (\varphi_1 \cup \varphi_2 \cup \{\mathtt{new}\,\pi\})}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau} \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau} \;\&\; \varphi'} \text{ if } \varphi \subseteq \varphi'$$

Table 3. Side Effect Analysis: $\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau} \;\&\; \varphi$ (Examples 5, 6 and 7).

the rules we make sure to record that a location at the relevant program point might have been created, referenced or assigned.

The purpose of ϱ in the rule for `new` in Table 3, and the purpose of the last rule in Table 3, is to ensure that we obtain a conservative extension of the underlying type system. The last rule is called a *subeffecting* rule and is essential in the presence of conditionals. The notation $\varphi \subseteq \varphi'$ means that φ is "a subset" of φ' (modulo *UCAI*). □

Example 6. Subtyping for Side Effect Analysis.

The last rule in Table 3 can be augmented with a rule for *subtyping*:

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau} \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau}' \;\&\; \varphi} \text{ if } \widehat{\tau} \leq \widehat{\tau}'$$

The ordering $\widehat{\tau} \leq \widehat{\tau}'$ on annotated types is derived from the ordering on annotations as follows:

$$\widehat{\tau} \leq \widehat{\tau} \qquad \frac{\widehat{\tau}_1' \leq \widehat{\tau}_1 \quad \widehat{\tau}_2 \leq \widehat{\tau}_2' \quad \varphi \subseteq \varphi'}{\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \leq \widehat{\tau}_1' \xrightarrow{\varphi'} \widehat{\tau}_2'} \qquad \frac{\widehat{\tau} \leq \widehat{\tau}' \quad \widehat{\tau}' \leq \widehat{\tau} \quad \varrho \subseteq \varrho'}{\widehat{\tau}\ \mathtt{ref}\ \varrho \leq \widehat{\tau}'\ \mathtt{ref}\ \varrho'}$$

Here $\varphi \subseteq \varphi'$ means that φ is "a subset" of φ' (modulo *UCAI*) and similarly $\varrho \subseteq \varrho'$ means that ϱ is "a subset" of ϱ' (modulo *UCAI*); as before $\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2$ is

contravariant in $\widehat{\tau}_1$ but *covariant* in φ and $\widehat{\tau}_2$. Also $\widehat{\tau}$ `ref` ϱ is both covariant in $\widehat{\tau}$ (when the reference variable is used for accessing its value as in $!x$) and contravariant in $\widehat{\tau}$ (when the reference variable is used for assignments as in x `:=` $\cdots$) whereas it is only covariant in ϱ. This form of subtyping amounts to *shape conformant subtyping* because $\widehat{\tau}_1 \leq \widehat{\tau}_2$ implies that the two annotated types have the same underlying types. □

Subeffecting alone suffices for obtaining a conservative extension of the underlying type system – provided that we regard the use of ϱ in the rule for `new` as being an integral part of subeffecting; the general idea is that subeffecting allows to "enlarge" the effects at an *early* point so that they do not conflict with the demands of the type and effect system. This reduces the usefulness of the effects but by incorporating subtyping we can "enlarge" the types at a *later* point; hence more informative types and effects can be used in subprograms. Coming back to our treatment of control flow analysis in Example 2 we note that basically it is a subeffecting analysis.

Polymorphism and Polymorphic Recursion

Subtyping is one of the classical techniques for making a type more useful by allowing to adapt it to different needs. Another classical technique is Hindley/Milner polymorphism as found in Standard ML and other functional languages. Both techniques are useful for increasing the precision of the information obtainable from type and effect systems.

Example 7. Polymorphism for Side Effect Analysis.

We now once more extend the language of Examples 5 and 6 with a polymorphic `let`-construct:

$$e ::= \cdots \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

We also allow types to contain type variables α, effects to contain annotation variables β and regions to contain region variables ρ:

$$\widehat{\tau} ::= \cdots \mid \alpha \qquad \varphi ::= \cdots \mid \beta \qquad \varrho ::= \cdots \mid \rho$$

We can then define type schemes: a type scheme is a type where a (possible empty) list $\zeta_1, \cdots, \zeta_n$ of type, effect and region variables has been quantified over:

$$\widehat{\sigma} ::= \forall(\zeta_1, \cdots, \zeta_n).\widehat{\tau}$$

If the list is empty we simply write $\widehat{\tau}$ for $\forall().\widehat{\tau}$.

The typing judgements will be of the form $\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\sigma}\ \&\ \varphi$ where the type environment $\widehat{\Gamma}$ now maps variables to type schemes (or types) and $\widehat{\sigma}$ is a type

scheme (or type). The clauses are as in Table 3 with the addition of the following rules:

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e_1 : \widehat{\sigma}_1 \ \& \ \varphi_1 \quad \widehat{\Gamma}[x \mapsto \widehat{\sigma}_1] \vdash_{\mathsf{SE}} e_2 : \widehat{\tau}_2 \ \& \ \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{SE}} \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \widehat{\tau}_2 \ \& \ \varphi_1 \cup \varphi_2}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau} \ \& \ \varphi}{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \forall(\zeta_1, \cdots, \zeta_n).\widehat{\tau} \ \& \ \varphi} \quad \text{if } \zeta_1, \cdots, \zeta_n \text{ do not occur free in } \widehat{\Gamma} \text{ and } \varphi$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \forall(\zeta_1, \cdots, \zeta_n).\widehat{\tau} \ \& \ \varphi}{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : (\theta\ \widehat{\tau}) \ \& \ \varphi} \quad \text{if } dom(\theta) \subseteq \{\zeta_1, \cdots, \zeta_n\}$$

The second and third rules are responsible for the polymorphism and are extensions of the last two rules of Table 2. The second rule is the *generalisation rule*: we can quantify over any type, annotation or region variable that does not occur free in the assumptions *or in the effect*. The third rule is the *instantiation rule*: we just apply a substitution in order to replace the bound type, annotation and region variables with other types, annotations and regions. □

Both subtyping and polymorphism improve subeffecting by giving finer control over when to "enlarge" the types; we already explained the advantage: that more informative types and effects can be used in subprograms. Since the mechanisms used are incomparable it clearly makes sense to combine both. However, as discussed in Section 5, it may be quite challenging to develop a type and effect inference algorithm that is both syntactically sound and complete.

Example 8. Region Inference.

The `let`-construct can be used to give polymorphic types to functions. But in the Hindley/Milner approach a recursive function can only be used polymorphically outside of its own body – inside its own body it must be used monomorphically. The generalisation to allow recursive functions to be used polymorphically also inside their own bodies is known as *polymorphic recursion* but gives rise to an undecidable type system; this means that no terminating type inference algorithm can be both syntactically sound and complete. This insight is a useful illustration of the close borderline between decidability and undecidability that holds for the inference based approach to the static analysis of programs.

Even though we abstain from using polymorphic recursion for ordinary types there is still the possibility of using polymorphic recursion for the effects annotating the ordinary and possibly polymorphic types given to recursive functions. In this way, distinct uses of a recursive function inside its body can still be analysed in different ways. This approach is taken in an analysis known as region inference [44] that is used when implementing functional languages in a stack-based regime rather than a heap-based regime. More precisely, the memory model is a stack of regions of data items, and the analysis facilitates determining at compile-time in which region to allocate data and when to deallocate a region (rather than using a garbage collector at run-time).

The use of polymorphic recursion for effect and region annotations allows the inference system to deal precisely with the allocation of data inside recursive functions. Furthermore, the inference system implicitly incorporates a notion of constraint between annotation variables and their meaning (via a dot notation on function arrows); as discussed in Section 5 this is a common feature of systems based on subtyping as otherwise principal types may not be expressible. To obtain effects that are as small as possible, the inference system uses "effect masking" [21, 39, 40] for removing internal components of the effect: effect components that only deal with regions that are not externally visible. It is unclear whether or not this system is decidable but nonetheless it has proved quite useful in practice: a syntactically sound inference algorithm has been devised and it is sufficiently accurate that a region-based implementation of Standard ML has turned out to compete favourably with a heap-based implementation. □

Mutually Recursive Types and Effects

So far the annotations and effects have not included any type information; as we shall see in Section 5 this is essential for being able to develop type and effect inference algorithms using a two-stage approach where first the types are determined and next the effects annotating the types. It is possible to be more permissive in allowing effects to contain type information and in allowing even the shape of types and type schemes to be influenced by the type information contained in the effects; as will be explained in Section 5 this calls for a more complex one-stage approach to type and effect inference algorithms.

Example 9. Polymorphic Typing in Standard ML.

The Hindley/Milner approach to polymorphism was originally conceived only for pure functional languages. Extending it to deal with side effects in the form of reference variables has presented quite a few obstacles. As an example consider the following program fragment in an ML-like language:

```
let x = new nil in (··· x:=cons(7,x) ··· x:=cons(true,x) ···)
```

Here `x` is declared as a new cell whose contents is initially the empty list `nil` and it might be natural to let the type of `x` be something like $\forall\alpha.(\alpha\,\texttt{list})\,\texttt{ref}$; but then both assignments will typecheck and hence the type system will be semantically unsound as Standard ML only permits homogeneous lists where all elements have the same type.

Several systems have been developed for overcoming these problems (see e.g. [42]). One approach is to restrict the ability to generalise over "imperative" type variables: these are the type variables that may be used in an imperative manner. It is therefore natural to adapt the side effect analysis to record the imperative type variables and to prohibit the generalisation rule from generalising over imperative type variables. In this way the shape of type schemes is clearly influenced

by the effect information. This idea occurred already in [39, 40, 48] in the form of an extended side effect analysis with polymorphism and subeffecting. □

4 Causal Type Systems

So far the annotations and effects have had a rather simple structure in that they have mainly been sets. It is possible to be more ambitious in identifying the "causality" or temporal order among the various operations. As an example, we now consider the task of extracting *behaviours* (reminiscent of terms in a process algebra) from programs in Concurrent ML by means of a type and effect system; here effects (the behaviours) have structure, they may influence the type information (as in Example 9), and there are inference rules for subeffecting and shape conformant subtyping. These ideas first occurred in [27, 29] (not involving polymorphism) and in [2, 33, 34] (involving polymorphism); our presentation is mainly based on [33, 34] because the inference system is somewhat simpler than that of [1] (at the expense of making it harder to develop an inference algorithm); we refer to [1, Chapter 1] for an overview of some of the subtle technical details. An application to the validation of embedded systems is presented in [32] where a control program is shown not to satisfy the safety requirements.

Example 10. Adding Constructs for Communication.

To facilitate the communication analysis we shall add constructs for creating new channels, for generating new processes, and for communicating between processes over typed channels:

$$e ::= \cdots \mid \texttt{channel}_\pi \mid \texttt{spawn}\ e_0 \mid \texttt{send}\ e_1\ \texttt{on}\ e_2 \mid \texttt{receive}\ e_0$$

Here $\texttt{channel}_\pi$ creates a new channel identifier, $\texttt{spawn}\ e_0$ generates a new parallel process that executes e_0, and $\texttt{send}\ v\ \texttt{on}\ ch$ sends the value v to another process ready to receive a value by means of $\texttt{receive}\ ch$. We shall assume that there is a special constant `()` of type `unit`; this is the value to be returned by the `spawn` and `send` constructs. □

Example 11. Communication Analysis.

Turning to the communication analysis the annotations of interest are:

$$\begin{aligned}
\varphi &::= \beta \mid \Lambda \mid \varphi_1;\varphi_2 \mid \varphi_1 + \varphi_2 \mid \mathsf{rec}\beta.\varphi \mid \widehat{\tau}\ \mathsf{chan}\ \varrho \mid \mathsf{spawn}\ \varphi \mid \varrho!\widehat{\tau} \mid \varrho?\widehat{\tau} \\
\varrho &::= \rho \mid \{\pi\} \mid \varrho_1 \cup \varrho_2 \mid \emptyset \\
\widehat{\tau} &::= \alpha \mid \texttt{int} \mid \texttt{bool} \mid \cdots \mid \texttt{unit} \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \mid \widehat{\tau}\ \texttt{chan}\ \varrho \\
\widehat{\sigma} &::= \forall(\zeta_1, \cdots, \zeta_n).\widehat{\tau}
\end{aligned}$$

The behaviour Λ is used for atomic actions that do not involve communication; in a sense it corresponds to the empty set in previous annotations although it

$$\widehat{\Gamma} \vdash_{\mathsf{COM}} c : \widehat{\tau}_c \;\&\; \Lambda \qquad\qquad \widehat{\Gamma} \vdash_{\mathsf{COM}} x : \widehat{\Gamma}(x) \;\&\; \Lambda$$

$$\frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_x] \vdash_{\mathsf{COM}} e_0 : \widehat{\tau}_0 \;\&\; \varphi_0}{\widehat{\Gamma} \vdash_{\mathsf{COM}} \mathtt{fn}_\pi\ x\ \mathtt{=>}\ e_0 : \widehat{\tau}_x \xrightarrow{\varphi_0} \widehat{\tau}_0 \;\&\; \Lambda}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{COM}} e_1 : \widehat{\tau}_2 \xrightarrow{\varphi_0} \widehat{\tau}_0 \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{COM}} e_2 : \widehat{\tau}_2 \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{COM}} e_1\ e_2 : \widehat{\tau}_0 \;\&\; \varphi_1; \varphi_2; \varphi_0}$$

$$\widehat{\Gamma} \vdash_{\mathsf{COM}} \mathtt{channel}_\pi : \widehat{\tau}\ \mathtt{chan}\ \{\pi\} \;\&\; \widehat{\tau}\ \mathtt{chan}\ \{\pi\}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{COM}} e_0 : \widehat{\tau}_0 \;\&\; \varphi_0}{\widehat{\Gamma} \vdash_{\mathsf{COM}} \mathtt{spawn}\ e_0 : \mathtt{unit} \;\&\; \mathtt{spawn}\ \varphi_0}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{COM}} e_1 : \widehat{\tau} \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{COM}} e_2 : \widehat{\tau}\ \mathtt{chan}\ \varrho_2 \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{COM}} \mathtt{send}\ e_1\ \mathtt{on}\ e_2 : \mathtt{unit} \;\&\; \varphi_1; \varphi_2; (\varrho_2 ! \widehat{\tau})}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{COM}} e_0 : \widehat{\tau}\ \mathtt{chan}\ \varrho_0 \;\&\; \varphi_0}{\widehat{\Gamma} \vdash_{\mathsf{COM}} \mathtt{receive}\ e_0 : \widehat{\tau} \;\&\; \varphi_0; (\varrho_0 ? \widehat{\tau})}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{COM}} e : \widehat{\tau} \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{COM}} e : \widehat{\tau}' \;\&\; \varphi'} \quad \text{if } \widehat{\tau} \leq \widehat{\tau}' \text{ and } \varphi \sqsubseteq \varphi'$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{COM}} e : \widehat{\tau} \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{COM}} e : \forall(\zeta_1, \cdots, \zeta_n).\widehat{\tau} \;\&\; \varphi} \quad \text{if } \zeta_1, \cdots, \zeta_n \text{ do not occur free in } \widehat{\Gamma} \text{ and } \varphi$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{COM}} e : \forall(\zeta_1, \cdots, \zeta_n).\widehat{\tau} \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{COM}} e : (\theta\ \widehat{\tau}) \;\&\; \varphi} \quad \text{if } dom(\theta) \subseteq \{\zeta_1, \cdots, \zeta_n\}$$

Table 4. Communication Analysis: $\widehat{\Gamma} \vdash_{\mathsf{COM}} e : \widehat{\tau} \;\&\; \varphi$ (Example 11).

will be more intuitive to think of it as the empty string in regular expressions or as the silent action in process calculi. The behaviour $\varphi_1; \varphi_2$ says that φ_1 takes place before φ_2 whereas $\varphi_1 + \varphi_2$ indicates a choice between φ_1 and φ_2; this is reminiscent of constructs in regular expressions as well as in process algebras. The construct $\mathsf{rec}\beta.\varphi$ indicates a recursive behaviour that acts as given by φ except that any occurrence of β stands for $\mathsf{rec}\beta.\varphi$ itself.

The behaviour $\widehat{\tau}\ \mathsf{chan}\ \varrho$ indicates that a new channel has been allocated for communicating entities of type $\widehat{\tau}$; the region ϱ indicates the set of program points $\{\pi_1, \cdots, \pi_n\}$ where the creation could have taken place. The behaviour $\mathsf{spawn}\ \varphi$ indicates that a new process has been generated and that it operates as described by φ. The construct $\varrho ! \widehat{\tau}$ indicates that a value is sent over a channel of type $\widehat{\tau}\ \mathsf{chan}\ \varrho$, and $\varrho ? \widehat{\tau}$ indicates that a value is received over a channel of that type; this is reminiscent of constructs in most process algebras (in particular CSP).

The typing judgements have the form $\widehat{\Gamma} \vdash_{\mathsf{COM}} e : \widehat{\sigma}$ & φ where the type environment $\widehat{\Gamma}$ maps variables to type schemes (or types), $\widehat{\sigma}$ is the type scheme (or type) for the expression e, and φ is the behaviour that may arise during evaluation of e. The analysis is specified by the axioms and rules of Tables 4 and have many points in common with those we have seen before; we explain the differences below.

The axioms for constants and variables differ from the similar axioms in Table 3 in that Λ is used instead of $\emptyset$. A similar remark holds for the rule for function abstraction. In the rule for function application we now use sequencing to express that we first evaluate the function part, then the argument and finally the body of the function; note that the left-to-right evaluation order is explicit in the behaviour.

The axiom for channel creation makes sure to record the program point in the type as well as the behaviour, the rule for spawning a process encapsulates the behaviour of the spawned process in the behaviour of the construct itself and the rules for sending and receiving values over channels indicate the order in which the arguments are evaluated and then produce the behaviour for the action taken. The rules for generalisation and instantiation are much as before.

The rule for *subeffecting* and *subtyping* is an amalgamation of the rules in Table 3 and Example 6. Also note that there is no ϱ in the axiom for `channel` unlike in the axiom for `new` in Table 3; this is because the presence of subtyping makes it redundant. The ordering $\widehat{\tau} \leq \widehat{\tau}'$ on types is given by

$$\widehat{\tau} \leq \widehat{\tau} \qquad \frac{\widehat{\tau}_1' \leq \widehat{\tau}_1 \quad \widehat{\tau}_2 \leq \widehat{\tau}_2' \quad \varphi \sqsubseteq \varphi'}{\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \leq \widehat{\tau}_1' \xrightarrow{\varphi'} \widehat{\tau}_2'} \qquad \frac{\widehat{\tau} \leq \widehat{\tau}' \quad \widehat{\tau}' \leq \widehat{\tau} \quad \varrho \subseteq \varrho'}{\widehat{\tau}\ \mathtt{chan}\ \varrho \leq \widehat{\tau}'\ \mathtt{chan}\ \varrho'}$$

and is similar to the definition in Example 6: $\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2$ is contravariant in $\widehat{\tau}_1$ but covariant in φ and $\widehat{\tau}_2$, and $\widehat{\tau}\ \mathtt{chan}\ \varrho$ is both covariant in $\widehat{\tau}$ (for when a value is sent) and contravariant in $\widehat{\tau}$ (for when a value is received) and it is covariant in ϱ. As before, the ordering $\varrho \subseteq \varrho'$ means that ϱ is "a subset of" of ϱ' (modulo *UCAI*). However, the ordering $\varphi \sqsubseteq \varphi'$ on behaviours is more complex than what has been the case before because of the rich structure possessed by behaviours. The definition is given in Table 5 and will be explained below. Since the syntactic categories of types and behaviours are mutually recursive also the definitions of $\widehat{\tau} \leq \widehat{\tau}'$ and $\varphi \sqsubseteq \varphi'$ need to be interpreted recursively.

The axiomatisation of $\varphi \sqsubseteq \varphi'$ ensures that we obtain a preorder that is a congruence with respect to the operations for combining behaviours. Furthermore, sequencing is an associative operation with Λ as identity and we have a distributive law with respect to choice. It follows that choice is associative and commutative. Next the axioms for recursion allow us to unfold the rec-construct. The final three rules clarify how behaviours depend upon types and regions: $\widehat{\tau}\ \mathsf{chan}\ \varrho$ is both contravariant and covariant in $\widehat{\tau}$ and is covariant in ϱ (just as was the case for the type $\widehat{\tau}\ \mathtt{chan}\ \varrho$); $\varrho!\widehat{\tau}$ is covariant in both ϱ and $\widehat{\tau}$ (because a value is sent) whereas $\varrho?\widehat{\tau}$ is covariant in ϱ and contravariant in $\widehat{\tau}$ (because a value is

$\varphi \sqsubseteq \varphi$	$\dfrac{\varphi_1 \sqsubseteq \varphi_2 \quad \varphi_2 \sqsubseteq \varphi_3}{\varphi_1 \sqsubseteq \varphi_3}$
$\dfrac{\varphi_1 \sqsubseteq \varphi_2 \quad \varphi_3 \sqsubseteq \varphi_4}{\varphi_1;\varphi_3 \sqsubseteq \varphi_2;\varphi_4}$	$\dfrac{\varphi_1 \sqsubseteq \varphi_2 \quad \varphi_3 \sqsubseteq \varphi_4}{\varphi_1+\varphi_3 \sqsubseteq \varphi_2+\varphi_4}$
$\varphi_1;(\varphi_2;\varphi_3) \sqsubseteq (\varphi_1;\varphi_2);\varphi_3$	$(\varphi_1;\varphi_2);\varphi_3 \sqsubseteq \varphi_1;(\varphi_2;\varphi_3)$
$\varphi \sqsubseteq \Lambda;\varphi \qquad \Lambda;\varphi \sqsubseteq \varphi$	$\varphi \sqsubseteq \varphi;\Lambda \qquad \varphi;\Lambda \sqsubseteq \varphi$
$(\varphi_1+\varphi_2);\varphi_3 \sqsubseteq (\varphi_1;\varphi_3)+(\varphi_2;\varphi_3)$	$(\varphi_1;\varphi_3)+(\varphi_2;\varphi_3) \sqsubseteq (\varphi_1+\varphi_2);\varphi_3$
$\varphi_1 \sqsubseteq \varphi_1+\varphi_2 \quad \varphi_2 \sqsubseteq \varphi_1+\varphi_2$	$\varphi+\varphi \sqsubseteq \varphi$
$\dfrac{\varphi_1 \sqsubseteq \varphi_2}{\mathsf{spawn}\ \varphi_1 \sqsubseteq \mathsf{spawn}\ \varphi_2}$	$\dfrac{\varphi_1 \sqsubseteq \varphi_2}{\mathsf{rec}\beta.\varphi_1 \sqsubseteq \mathsf{rec}\beta.\varphi_2}$
$\mathsf{rec}\beta.\varphi \sqsubseteq \varphi[\beta \mapsto \mathsf{rec}\beta.\varphi]$	$\varphi[\beta \mapsto \mathsf{rec}\beta.\varphi] \sqsubseteq \mathsf{rec}\beta.\varphi$
$\dfrac{\varrho_1 \subseteq \varrho_2 \quad \widehat{\tau}_1 \leq \widehat{\tau}_2}{\varrho_1!\widehat{\tau}_1 \sqsubseteq \varrho_2!\widehat{\tau}_2}$	$\dfrac{\varrho_1 \subseteq \varrho_2 \quad \widehat{\tau}_2 \leq \widehat{\tau}_1}{\varrho_1?\widehat{\tau}_1 \sqsubseteq \varrho_2?\widehat{\tau}_2}$
$\dfrac{\widehat{\tau} \leq \widehat{\tau}' \quad \widehat{\tau}' \leq \widehat{\tau} \quad \varrho \subseteq \varrho'}{\widehat{\tau}\ \mathsf{chan}\ \varrho \sqsubseteq \widehat{\tau}'\ \mathsf{chan}\ \varrho'}$	

Table 5. Ordering on behaviours: $\varphi \sqsubseteq \varphi'$ (Example 11).

received). There is no explicit law for renaming bound behaviour variables as we shall regard $\mathsf{rec}\beta.\varphi$ as being equal to $\mathsf{rec}\beta'.\varphi'$ when they are α-equivalent. □

5 The Methodology

So far we have illustrated the variety of type and effect systems that can be found in the literature. Now we turn to explaining the individual steps in the overall methodology of designing and using type and effect systems:

- devise a semantics for the programming language,
- develop a program analysis in the form of a type and effect system (this is what Sections 2, 3 and 4 have given numerous examples of),
- prove the semantic correctness of the analysis,
- develop an efficient inference algorithm,
- prove that the inference algorithm is syntactically sound and complete, and
- utilise the information for applications like program transformations or improved code generation.

Each of these phases have their own challenges and open problems that we now consider in some detail; many of these issues are rather orthogonal to the

more syntactic differences used to distinguish between the formulations used in Sections 2, 3 and 4.

Semantics. Semantics is a rather well understood area. In principle both denotational and operational semantics can be used as the foundations for type and effect systems but most papers in the literature take an operational approach. This is indeed very natural when the analysis needs to express further intensional details than are normally captured by a denotational semantics. But even when taking an operational approach one frequently needs to devise it in such a manner that it captures those operational details for which the analysis is intended. The term *instrumented semantics* [17] has been coined for a class of denotational or operational semantics that are more precise about low-level machine detail (say concerning pipe-lining or the number and nature of registers) than usual. It is therefore wise to be cautious about the precise meaning of claims stating that an analysis has been proved correct with respect to "the" semantics.

The inference system. Previous sections have illustrated some of the variations possible when developing type and effect systems as well as some of the applications for which they can be used. However, it would be incorrect to surmise that the selection of components are inherently linked to the example analysis where they were first illustrated.

At the same time we illustrated a number of design considerations to be taken into account when devising a type and effect system. In our view the major design decisions are as follows:

- whether or not to incorporate
  - subeffecting,
  - subtyping,
  - polymorphism, and
  - polymorphic recursion,
- whether or not types are allowed to be influenced by effects (as was the case in Example 9 and Section 4), and
- whether or not constraints are an explicit part of the inference system (unlike what simplicity demanded us to do here).

The choices made have a strong impact on the difficulties of obtaining a syntactically sound and complete inference algorithm; indeed, for some combinations it may be beyond state-of-the-art (or even impossible) and in particular it may be hard (or impossible) to deal with subtyping without admitting constraints to the inference system. An important area of further research is how to identify those features of the annotated type and effect systems that lead to algorithmic intractability.

Often the type and effect system is developed for a typed programming language. It is then important to ensure that whenever a program can be typed in the

original type system then there also exists a type in the type and effect system, and whenever there exists a type in the type and effect system then the program can also be typed in the original type system. This is established by proving that the type and effect system is a *conservative extension* of the original or underlying type system. It is also possible to investigate whether or not the type and effect system admits principal types and effects; luckily this will always be the case if a syntactically sound and complete inference algorithm can be developed.

Further studies are needed to understand the interplay between type and effect systems and the other approaches to static analysis of programs. It is interesting to note that the existence of principal types is intimately connected to the notion of Moore families used in abstract interpretation: a principal type roughly corresponds to the least solution of an equation system.

Semantic correctness. Many of the techniques needed for establishing semantic soundness (sometimes called type soundness) are rather standard. For operational semantics the statement of correctness generally take the form of a *subject reduction result*: if a program e has a type τ and if e evaluates to e' then also e' has the type τ; this approach to semantic correctness has a rather long history [24, 25, 49] and applies both to small-step Structural Operational Semantics and to big-step Natural Semantics [36]. It is important to stress that the correct use of covariance and contravariance (in the rules for subtyping) is essential for semantic correctness to hold.

For more complex situations the formulation of "has a type" may have to be defined coinductively [42], in which case also the proof of the subject reduction result may need to exploit coinduction (e.g. [30]), and the notions of Kripke relations and Kripke-logical relations (see e.g. [28]) may be useful when using a denotational semantics [26]. We refer to [3, 39, 40, 45] for a number of applications of these techniques.

The inference algorithm. The development of a syntactically sound and complete inference algorithm may be based on the ideas in [20, 41]. The simplest approach is a *two-stage approach* where one first determines the underlying types and next determines the (possibly polymorphic) effects on top of the explicitly typed programs. The basic idea is to ensure that the type inference algorithm operates on a *free algebra* by restricting annotations to be annotation variables only (the concept of "*simple types*") and by recording a set of constraints for the meaning of the annotation variables. This suffices for adapting the established techniques for polymorphic type inference, by means of the classical algorithm $\mathcal{W}$ developed in [6, 22] for Hindley/Milner polymorphism, to the setting at hand. In this scheme one might have $\mathcal{W}(\Gamma, e) = (S, \tau, \varphi, C)$ where e is the program to be typed, τ is the form of the resulting type and φ summarises the overall effect of the program. In case e contains free variables we need preliminary information

about their types and this is provided by the type environment Γ; as a result of the type inference this preliminary information may need to be modified as reported in the substitution S. Finally, C is a set of constraints that record the meaning of the annotation variables. For efficiency the algorithmic techniques often involve the generation of constraint systems in a program independent representation.

In the case of polymorphic recursion decidability becomes an issue. Indeed, polymorphic recursion over type variables makes the polymorphic type system undecidable. It is therefore wise to restrict the polymorphic recursion to annotation variables only as in [44]. There the first stage is still ordinary type inference; the second stage [43] concerns an algorithm $\mathcal{S}$ that generates effect and region variables and an algorithm $\mathcal{R}$ that deals with the complications due to polymorphic recursion (for effects and regions only). The inference algorithm is proved syntactically sound but is known not to be syntactically complete; indeed, obtaining an algorithm that is syntactically sound as well as complete, seems beyond state-of-the-art.

Once types and effects are allowed to be mutually recursive, the two-stage approach no longer works for obtaining an inference algorithm because the effects are used to control the shape of the underlying types (in the form of which type variables are included in a polymorphic type). This suggests a *one-stage approach* where special care needs to be taken when deciding the variables over which to generalise when constructing a polymorphic type. The main idea is that the algorithm needs to consult the constraints in order to determine a larger set of forbidden variables than those directly occurring in the type environment or the effect; this can be formulated as a *downwards closure* with respect to the constraint set [31, 48] or by taking a *principal solution* of the constraints into account [39, 40].

Adding subtyping to this development dramatically increases the complexity of the development. The integration of shape conformant subtyping, polymorphism and subeffecting is done in [3, 31, 35] that develop an inference algorithm that is proved syntactically sound; these papers aimed at integrating the techniques for polymorphism and subeffecting (but no subtyping) from effect systems [39, 40, 48] with the techniques for polymorphism and subtyping (but no effects) from type systems [16, 37, 38]. A more ambitious development where the inference system is massaged so as to facilitate developing an inference algorithm that is also syntactically complete is described in [1]; the inference system used there has explicit constraints in the inference system (as is usually the case in type systems based on subtyping).

Syntactic soundness and completeness. The syntactic soundness and completeness results to be established present a useful guide to developing the inference algorithm. Formulations of syntactic soundness are mostly rather straightforward: the result produced by the algorithm must give rise to a valid inference

in the inference system. A simple example is the following: if $\mathcal{W}(\Gamma, e) = (S, \tau, \varphi)$ then $S(\Gamma) \vdash e : \tau \,\&\, \varphi$ must hold; here it is clear that the substitution produced is intended to refine the initial information available when first calling the algorithm. A somewhat more complex example is: if $\mathcal{W}(\Gamma, e) = (S, \tau, \varphi, C)$ then $S'(S(\Gamma)) \vdash e : S'(\tau) \,\&\, S'(\varphi)$ must hold whenever S' is a solution to the constraints in C. The proofs are normally by structural induction on the syntax of programs.

The formulations of syntactic completeness are somewhat more involved. Given a program e such that $\Gamma_\diamond \vdash e : \tau_\diamond \,\&\, \varphi_\diamond$, the main difficulty is to show how this can be obtained from $\mathcal{W}(\Gamma, e) = (S, \tau, \varphi)$ or $\mathcal{W}(\Gamma, e) = (S, \tau, \varphi, C)$. The solution is to formally define when one "typing" is an instance of another; the notion of *lazy instance* [9] is very useful here and in more complex scenarios Kripke-logical relations (see e.g. [28]) may be needed [1]. The proofs are often challenging and often require developing extensive techniques for "normalising" deductions made in the inference system so as to control the use of non-syntax directed rules. For sufficiently complex scenarios syntactic completeness may fail or may be open (as mentioned above); luckily soundness often suffices for the inference algorithm to be of practical use.

Exploitation. Exploitation is a rather open-ended area although it would seem that the integration of program analyses and program transformations into an inference based formulation is quite promising [46]. Indeed, inference-based formulations of analyses can be seen as an abstract counterpart of the use of attribute grammars when developing analyses in compilers, and in the same way inference-based formulations of analyses and transformations can be seen as an abstract counterpart of the use of attributed transformation grammars [47].

6 Conclusion

The approach based on type and effect systems is a promising approach to the static analysis of programs because the usefulness of types has already been widely established. The main strength lies in the ability to interact with the user: clarifying what the analysis is about (and when it may fail to be of any help) and in propagating the results back to the user in an understable way (which is not always possible for flow based approaches working on intermediate representations). The main areas of further research concern the expressiveness of the inference based specifications, the complexity and decidability of the inference algorithms and the interplay with the other approaches to static analysis of programs.

Acknowledgements. We wish to thank Torben Amtoft for working with us for many years on type and effect systems; we would also like to thank the referees for their careful reading and helpful comments.

References

1. T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.
2. T. Amtoft, F. Nielson, and H.R. Nielson. Type and behaviour reconstruction for higher-order concurrent programs. *Journal of Functional Programming*, 7(3):321–347, 1997.
3. T. Amtoft, F. Nielson, H.R. Nielson, and J. Ammann. Polymorphic subtyping for effect analysis: The dynamic semantics. In *Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 172–206. Springer, 1997.
4. P. N. Benton. Strictness logic and polymorphic invariance. In *Proc. Second International Symposium on Logical Foundations of Computer Science*, volume 620 of *Lecture Notes in Computer Science*, pages 33–44. Springer, 1992.
5. P. N. Benton. Strictness properties of lazy algebraic datatypes. In *Proc. WSA '93*, volume 724 of *Lecture Notes in Computer Science*, pages 206–217. Springer, 1993.
6. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. POPL '82*, pages 207–212. ACM Press, 1982.
7. K.-F. Faxén. Optimizing lazy functional programs using flow inference. In *Proc. SAS '95*, volume 983 of *Lecture Notes in Computer Science*, pages 136–153. Springer, 1995.
8. K.-F. Faxén. Polyvariance, polymorphism, and flow analysis. In *Proc. Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 260–278. Springer, 1997.
9. Y.-C. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Proc. TAPSOFT '89*, volume 352 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1989.
10. Y.-C. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73:155–175, 1990.
11. N. Heintze. Control-flow analysis and type systems. In *Proc. SAS '95*, volume 983 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 1995.
12. Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with Secrecy and Integrity. In *Proc. POPL '98*, pages 365–377. ACM Press, 1998.
13. F. Henglein and C. Mossin. Polymorphic binding-time analysis. In *Proc. ESOP '94*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer, 1994.
14. T. P. Jensen. Strictness analysis in logical form. In *Proc. FPCA '91*, volume 523 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 1991.
15. T. P. Jensen. Disjunctive strictness analysis. In *Proc. LICS '92*, pages 174–185, 1992.
16. M. P. Jones. A theory of qualified types. In *Proc. ESOP '92*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 1992.
17. N. D. Jones and F. Nielson. Abstract Interpretation: a Semantics-Based Tool for Program Analysis. In *Handbook of Logic in Computer Science volume 4*. Oxford University Press, 1995.
18. P. Jouvelot. Semantic Parallelization: a practical exercise in abstract interpretation. In *Proc. POPL '87*, pages 39–48, 1987.
19. P. Jouvelot and D. K. Gifford. Reasoning about continuations with control effects. In *Proc. PLDI '89*, ACM SIGPLAN Notices, pages 218–226. ACM Press, 1989.

20. P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Proc. POPL '91*, pages 303–310. ACM Press, 1990.
21. J. M. Lucassen and D. K. Gifford. Polymorphic effect analysis. In *Proc. POPL '88*, pages 47–57. ACM Press, 1988.
22. R. Milner. A theory of type polymorphism in programming. *Journal of Computer Systems*, 17:348–375, 1978.
23. J. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.
24. F. Nielson. A formal type system for comparing partial evaluators. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Proc. Partial Evaluation and Mixed Computation*, pages 349–384. North Holland, 1988.
25. F. Nielson. The typed λ-calculus with first-class processes. In *Proc. PARLE'89*, volume 366 of *Lecture Notes in Computer Science*, pages 355–373. Springer, 1989.
26. F. Nielson and H. R. Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.
27. F. Nielson and H. R. Nielson. From CML to process algebras. In *Proc. CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 493–508. Springer, 1993.
28. F. Nielson and H. R. Nielson. Layered predicates. In *Proc. REX'92 workshop on Semantics — foundations and applications*, volume 666 of *Lecture Notes in Computer Science*, pages 425–456. Springer, 1993.
29. F. Nielson and H. R. Nielson. From CML to its process algebra. *Theoretical Computer Science*, 155:179–219, 1996.
30. F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.
31. F. Nielson, H.R. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: The algorithm. In *Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 207–243. Springer, 1997.
32. H. R. Nielson, T. Amtoft, and F. Nielson. Behaviour analysis and safety conditions: a case study in CML. In *Proc. FASE '98*, number 1382 in Lecture Notes in Computer Science, pages 255–269. Springer, 1998.
33. H. R. Nielson and F. Nielson. Higher-Order Concurrent Programs with Finite Communication Topology. In *Proc. POPL '94*. Springer, 1994.
34. H. R. Nielson and F. Nielson. Communication analysis for Concurrent ML. In F. Nielson, editor, *ML with Concurrency*, Monographs in Computer Science, pages 185–235. Springer, 1997.
35. H.R. Nielson, F. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In *Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 141–171. Springer, 1997.
36. G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Denmark, 1981.
37. G. S. Smith. Polymorphic inference with overloading and subtyping. In *Proc. TAPSOFT '93*, volume 668 of *Lecture Notes in Computer Science*, pages 671–685. Springer, 1993.
38. G. S. Smith. Polymorphic type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.
39. J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Proc. LICS '92*, pages 162–173, 1992.
40. J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

41. Y.-M. Tang. *Control-Flow Analysis by Effect Systems and Abstract Interpretation*. PhD thesis, Ecole des Mines de Paris, 1994.
42. M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.
43. M. Tofte and L. Birkedal. A region inference algorithm. *ACM TOPLAS*, 20(3):1–44, 1998.
44. M. Tofte and J.-P. Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proc. POPL '94*, pages 188–201. ACM Press, 1994.
45. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132:109–176, 1997.
46. M. Wand. Specifying the correctness of binding-time analysis. In *Proc. POPL '93*, pages 137–143, 1993.
47. R. Wilhelm. Global flow analysis and optimization in the MUG2 compiler generating system. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 5. Prentice Hall International, 1981.
48. A. K. Wright. Typing references by effect inference. In *Proc. ESOP '92*, volume 582 of *Lecture Notes in Computer Science*, pages 473–491. Springer, 1992.
49. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.