Algorithmic

Trace Effect Analysis

A Thesis Presented

by

David Van Horn

to

The Faculty of the Graduate College

of

The University of Vermont

In Partial Fullfillment of the Requirements
for the Degree of Master of Science
Specializing in Computer Science

May, 2006

Accepted by the Faculty of the Graduate College, The University of Vermont, in partial fulfillment of the requirements for the degree of Master of Science, specializing in Computer Science.

Thesis Examination Committee:

Christian Skalka, Ph.D.	Advisor
Abdullah Arslan, Ph.D.	
Craig Damon, Ph.D.	
Jun Yu, Ph.D.	Chairperson
Frances E. Carr, Ph.D.	Vice President for Research and Dean of the Graduate College

Date: March 29, 2006

Abstract

Trace effect analysis empowers programmers to make assertions on the temporal sequence of atomic program events having occurred at any point in the computation of a program. A polymorphic type and effect inference system automatically extracts an abstract interpretation conservatively approximating the events and assertions that will arise at run-time. Such an interpretation can then be model-checked to obtain a static verification of these temporal program logics for higher-order programs of a λ-calculus extended with notions of atomic events, temporal assertions, and computational *traces*.

The purpose of this thesis is to demonstrate that trace effect analysis is implementable and to prove the implementation sound with respect to its logical specification. This thesis describes both the logical and algorithmic type and effect system and applications of the analysis to the static enforcement of security mechanisms. The systems use an effectual weakening rule, and enjoy a unified representation of types resulting in concise specifications of programs. A type inference algorithm is presented and an algorithmic soundness result is established with respect to its logical counterpart. An implementations of the inference algorithm is provided. Finally, extensions to the system are defined, discussed, and implemented.

for Jessie

Acknowledgements

Three professors and their respective graduate seminars were vital to the development of this thesis. I would like to acknowledge first and foremost, Professor Skalka's seminar *Types in Programming Languages*, held in Spring 2004. The course introduced me to constraint type systems and their inference algorithms, as well as a number of type-theoretic concepts, which have shaped my development of algorithmic trace effect analysis. Professor Damon's seminar *Tractible Software Analysis* in Spring 2005 was also fundamental. It was in this course that I studied abstract interpretation and model checking, which have directly impacted this work. It was also in this course that I was able to present early results from this thesis and the feedback has been invaluable. Finally, Professor Arslan's Spring 2005 *Advanced Topics in Theory of Computation* allowed me to study complexity results for type reconstruction problems in the λ-calculus and helped develop my theorem-proving skills. I am grateful to these three teachers for accepting my invitation to sit on this thesis review committee.

I would also like to acknowledge the University of Vermont Computer Security Lab, which gave me the opportunity to study programming language-based approaches to security and present recent research in the field and the Computer Science Student Association, which allowed me to invite a number of prominent researchers to visit the university.

I am deeply indebted to Professor Skalka, my adviser, for creating such a rich scholarly environment in which to work. I thank him for garnering the financial support that made my graduate studies possible and I thank him for the intellectual support that made my studies compelling and fruitful.

None of this work could have been done without the support of my family and dearest friends. They have kept me from total madness.

Table of Contents

List of Figures

1 Introduction

Many program correctness properties are expressible as temporal properties of *program event traces*—the ordered sequence of steps taken during computation—such as the property that files are opened before being read, memory allocation occurs before its use, and access control properties such as privilege activation occurs before privileged action. Such well-formedness properties of traces are expressible and enforceable as program monitors or checks in program logics that occur at run-time (Schneider 2000). Checking these assertions incurs a run-time cost, and ill-formed computations will be caught only once the program executes. However, static program analyses, in the form of a programming language type system, allow for the conservative verification of these properties without running the program. Such an analysis was proposed in (Skalka and Smith 2004a), known as *trace effect analysis*, which uses a *type and effect* program abstraction, coupled with *model-checking* to enforce temporal properties statically.

This thesis demonstrates the analysis can be fully automated in the form of a type inference algorithm and that this algorithm soundly represents judgements in the analysis. Well-founded automation is crucial for writing robust and secure programs and our results contribute to the techniques of constructing reliable software.

1.1 Trace Effect Analysis

Trace effect analysis is a static programming language analysis for ensuring the correct behavior of software with respect to temporal properties of programs. These temporal properties express the well-formedness of program events, such as when files are opened or when resources are obtained. By enforcing program event order specifications, it can be guaranteed, for example, that files are opened before being read, or that privilege activation occurs before privileged resources are

acquired. Such well-formedness properties can describe certain security mechanisms and can be used to ensure handshake protocols are respected or to discover logical flaws in a program. As an example of the kind of properties that can be enforced under this analysis, consider the handshake protocols of secure socket layer (SSL). For a program sending and receiving data over an SSL socket, e.g. a web browser that supports `https`, the relevant events are opening and closing of sockets, and reading and writing of data packets. An example event trace produced by a program execution could be:

```
ssl_open("snork.cs.jhu.edu",socket_1);
ssl_hs_begin(socket_1);
ssl_hs_success(socket_1);
ssl_put(socket_1);
ssl_get(socket_1);
ssl_open("moo.cs.uvm.edu",socket_2);
ssl_hs_begin(socket_2);
ssl_put(socket_2);
ssl_close(socket_1);
ssl_close(socket_2)
```

According to the SSL protocol, for a socket to be read from or written to, the socket must first be opened, then a handshake must be initiated and confirmed successful. Only then can data be transfered over the socket. The above trace is illegal; data is put on `socket_2` before notification has been received that handshake was successful on that socket.

By virtue of being a *static* analysis, reasoning is performed at compile time, i.e. it takes place before program execution occurs, leading to the early detection of errors. Another consequence of such static reasoning is any program deemed correct by the analysis will never violate its specification at run time. Guarantees about program behavior hold in all possible circumstances of the program execution and such guarantees can be known *a priori*.

Common security mechanisms such as stack inspection which are found in the Java programming language and .NET framework are enforced via run time inspection of the sequence of events having occurred at the current point in time. However, these mechanisms are formalizable within trace effect analysis and thus can be enforced at compile time, implying all run time inspections are superfluous and can be eliminated. Any run time overhead associated with enforcing the security mechanism can safely be eliminated.

The analysis consists of a program logic (a system of axioms and deduction rules) such that if a program has a derivable logical judgment, the program meets its temporal specification. Moreover, this analysis can be automated by an inference algorithm automatically constructing such a proof for a program. This thesis proves the algorithm sound, which is to say any judgment found by the algorithm is a valid logical judgment and represents a proof that the program meets its specification. This proof discovery procedure has been implemented for a foundational programming language with notions of atomic events, temporal assertions, and computational *traces*. The programming language empowers programmers to make assertions on the temporal sequence of atomic program events having occurred at any point in the computation of a program. These assertions can be verified in a fully automated and sound manner.

This thesis provides algorithms realizing this analysis, given in the form of a type inference system, which is used to extract a model of the program's temporal behavior, and assertions over that behavior. These program logics for higher-order programs can be verified using model checking techniques and thus run-time checks are verified *a priori* and can safely be removed, eliminating all run-time penalties and facilitating the early detection of errors. Such a mechanical verification increases the reliability of programs, and contributes to sound approaches to the construction of reliable software.

1.2 Motivations

The primary motivation for this work is the need for approaches to construct reliable software. There is an increasing reliance by society upon a computing infrastructure, and yet software lacks the most basic assurances of its reliability. Since programming languages are the means of articulating computational artifacts, we employ the analytic method to gain assurance in the behavior of programs. The method allows us to reason about all possible executions of a program, and to prove that the program meets it's specification using a mechanical, automated process that is well-founded mathematically, thus obtaining trustworthy software. The automatic mechanism and establishment of its well-foundedness are a primary contribution of this thesis.

The need for soundly constructed and trustworthy software is apparent. The report "Information Technology Research: Investing in Our Future" (President's Information Technology Advisory Committee 1999) makes the case quite clear. The report states, "technologies to build reliable and secure software are inadequate," and goes on to state:

> Large software systems are beyond our capability to describe precisely. Consequently, there is little automation of their construction, little re-use of previously developed components, virtually no ability to perform accurate engineering analyses, and no way to know the extent to which a large software system has been tested.
>
> Having meaningful and standardized behavioral specifications would make it feasible to determine the properties of a software system and enable more thorough and less costly testing. Unfortunately such specifications are rarely used. Even less frequently is there a correspondence between a specification and the software itself. Often software behavior and flaws are observable only when the program is run, and even then may be invisible except under certain unusual conditions. Programs written in such circumstances frustrate attempts to create robust systems and are inherently fragile.
>
> [I]t has become clear that the processes of developing, testing, and maintaining software must change. We need scientifically sound approaches to software development that will enable meaningful and practical testing for consistency of specifications and implementations.

There have been significant advancements in "scientifically sound approaches to software

development" with respect to reliability and security, but there is much left to do, including the widespread adoption of such disciplines into everyday software development. Part of the problem is that sound approaches to software development *in general* have not yet received their due credit and adoption into mainstream development practices and programming languages. For example, language-based security is predicated on language safety, i.e programs behave only in a well-defined and consistent manner. Vulnerabilities caused in part by buffer overflows are a ubiquitous subject of alerts issued by CERT (United States Computer Emergency Readiness Team 2005), yet this problem has been solved by safe languages for decades.

Safety properties "assert that the system always stays within some allowed region" (Kupferman and Vardi 2001). A "safe language" typically refers to a language in which it is impossible to use data in an ill-defined way, and is therefore type and memory safe. Some languages, such as ML (Milner, Tofte, Harper, and MacQueen 1997), establish this safety through a typing discipline. That is, programs in ML can be statically analyzed to infer a type and a type safety result ensures that no well-typed program reaches an ill-formed state, and thus precludes many vulnerabilities for exploitation such as buffer overflow. The theory of λ_{trace} shares this theoretical technique of formulating a syntactic type safety result as done in ML (Wright and Felleisen 1994). However, compared to ML, type safety in this setting is a much richer notion; safety implies the satisfaction of all checks in a program and the typing discipline is extended to enforce this richer notion of safety.

The next two sections discuss and relate alternative approaches to reliable software construction. For these approaches, the strengths and weaknesses are contrasted and we describe how trace effect analysis synthesizes the best aspects and overcomes the weaknesses of these approaches.

1.3 Model Checking

Model checking has been put forth as an alternative to deductive methods in the verification of software correctness, advocated on the grounds that automation of verification effectively attenuates the high level of human guidance required of deductive methods. However, traditional approaches to model checking require the formulation and extraction of a model of a program that sufficiently captures the behavior of the program with respect to some domain of behavior that is of interest. Although model checking can be fully automated, model extraction traditionally is not. Model extraction requires significant human guidance, which is precisely what model checking sought to avoid—worse, it is highly error-prone and, unlike deductive methods that can be mechanically checked after construction, the extraction of an incorrect model can result in unsound security judgments. Thus the need for sound automatic extraction techniques is well documented in the model checking community (Holzmann and Smith 2001; Holzmann 2003).

The difficulty of correct model extraction arises from the semantic gap between real-world programming languages, such as Java, and the temporal logics employed by various verification tools. Thus, it is natural to turn to language based approaches to bridge this gap, as advocated above.

Beyond the model extraction problem, model checking can often be infeasible when establishing safety properties of programs that should apply to all possible executions of a program. As noted in (Kupferman and Vardi 2001):

> A computation that violates a general linear property reaches a bad cycle, which witnesses the violation of the property. Accordingly, current methods and tools for model checking of linear properties are based on a search for bad cycles. A symbolic implementation of such a search involves the calculation of a nested fixed-point expression over the system's state space, and is often infeasible.

Our approach is to use type inference to provide a means for the accurate and sound automatic extraction of a model from a program. Type safety results give "for all" guarantees about programs, and without a costly search of the system's state space. Type-based techniques such as subtyping/subeffecting and parametric polymorphism are applied to improve the accuracy of the extracted model, and thus combat the state explosion problem.

1.4 Abstract Interpretation

Abstract interpretation (Cousot and Cousot 1977; Jones and Nielson 1995; Nielson, Nielson, and Hankin 1999) is a theory of discrete approximation of the semantics of programming or specification languages. The fundamental idea of abstract interpretation is that programs are interpreted, not using their standard operational semantics, but rather over an abstract domain. As the domain becomes more abstract, the analysis becomes more tractable and less complete, but properties of the abstraction will always hold in the concrete semantics.

The basic idea of static program analysis is to use the computer to verify that a piece of software meets its specification—without actually executing the program. To do this, it is necessary to state a formal *computational model* that describes the operations executed during a computer's running of a program, including the internal effects and interactions with the environment in all possible conditions. The *semantics* of a program is the computational model describing the effective executions of the program in all possible environments and the *specification* is a computational model describing the desirable execution of the program in all possible environments. *Verification*, then, is a proof demonstrating that the semantics of the program is "contained" by the semantics of the specification.

Unfortunately, the static analysis of programs is hard, and for any nontrivial specification,

verification is intractable or undecidable. By considering only an approximation of possible run-time behaviors, it may be possible to regain tractability by compromising on precision—considering only a *sound*, *finite*, and *approximate* calculation of the programs' executions.

In abstract interpretation, two worlds are postulated, the concrete, and the abstract, along with mappings from sets of concrete values to the abstract value best describing the set, and from abstract values to sets of concrete values.

The relation between these two worlds is typically made precise by demonstrating the concrete semantics can be rebuilt from an abstraction of the program semantics,[1] and this rebuilt semantics should contain, at least, the original program semantics.[2] Thus properties that hold for the abstracted semantics, hold for the concrete semantics as well.

In this thesis, the concrete semantics of programs that are of interest are the *traces* of programs and the computational model is formulated as a one-step relation on machine configurations. The type and effect of a program are approximations of computation: the type approximates the set of values the program evaluates to should the program halt, and the effect approximates the set of possible traces generated at run-time. As such, this approximation is an abstract interpretation: it is a high-level abstraction of the control flow of a program (Schmidt and Steffen 1998). The type safety result shows that the static semantics of programs using trace effect analysis conservatively approximates that of the run-time semantics.

Much of the work on abstract interpretation has been done in the setting of imperative programs, first-order languages (that is, languages in which functions are not first-class values—they cannot be stored in a data structure, passed to or returned from functions, and so on), communicating sequential processes, parallel programs and logic programs.

[1] Known as a *Galois connection* (Melton, Schmidt, and Strecker 1986)
[2] Known as *near* commutativity.

There has been some work in abstract interpretation in higher order settings, starting with the work of Burn, Hankin, and Abramsky on strictness analysis (Burn, Hankin, and Abramsky 1986). Cousot and Cousot give an abstract interpretation of the simply typed, recursive λ calculus using a denotational semantics approach (Cousot and Cousot 1994). The interpretation generalizes strictness analysis, constant propagation, projection and PER-analysis. Shivers used abstract interpretation to develop control flow analysis for higher order languages using a similar denotational approach as Cousot and Cousot (Shivers 1991).

However, a small-step operational semantics approach, as employed in the paper, more naturally exposes the intermediate states of an abstract machine, vital for effectual analysis such as this. This approach has been taken by (Schmidt 1998b; Schmidt 1998a), however this was done in the setting of flowchart (first-order) programs.

1.5 Approach

This section develops the high level overview of the system before the more technical chapters that follow. This section introduces the type and effect approach, discusses the motivation behind a subeffecting system, and describes the subeffecting discipline adopted in this thesis: weakening. The section then includes a discussion of traditional type safety and how this concept is extended to realize a verification of temporal specifications. Finally, this section describes how this system improves upon and synthesizes the above approaches.

We take a *language-based approach*, integrating the necessary abstractions into a programming language so that a programmer can articulate temporal properties as part of the program, allowing the programmer to compose larger and more sophisticated abstractions using traditional programming notions.

Besides taking a language based approach, we take a *foundational approach* to studying event traces in a higher order setting, i.e. the analysis is developed in a λ-calculus setting, a primordial programming language representing the essence of higher order languages, such as functional and object-oriented programming languages. The foundational approach is taken so that the development of the analysis is feasible—the fundamental issues can be focused upon—and in order to study the analysis in the setting of the computational core underlying all programming languages. Further, the analysis of higher order programs has implications for both functional and object oriented languages.

1.5.1 Run-time traces

Our fundamental abstraction is an *event trace*, and our language is endowed with notions of *events*, and *checks*. An *event* is an abstract, atomic, program action, parameterized by a static constant. They are inserted by the programmer or compiler and serve to record some program action, such as opening a file, an access control privilege activation, or entry to or exit from a critical region. *Traces* are ordered sequences of events; whenever an event is encountered during program execution, it is appended to the current trace, and thus traces record the sequence of program events in temporal order. Since computations may be non-terminating, running for an arbitrarily long period, traces may be arbitrarily long sequences of events. A *check* is a predicate, expressed in a temporal logic, over arbitrarily long sequences of events. A check is also an event in that it is recorded in the computational trace, allowing policies to be expressed that assert properties about the order in which checks occur, too.

Our approach is to formulate a language model λ_{trace}, based on a λ-calculus[3] endowed with

[3]For a concise introduction and exposition of the lambda calculus, see (Hankin 2004). For an encyclopedic treatment, refer to (Barendregt 1984).

notions of atomic events and temporal assertions over events. The language is given a concrete, operational semantics. Reduction operates over a tuple of an expression being reduced and a trace accumulating the temporal order of events occurring throughout reduction. When a check expression occurs during computation, the logical formula is verified with respect to the current trace. The language remains abstract with respect to the syntax and semantics of checks, thus the meaning of a formula must be given in terms of a meaning function that gives true when the formula is satisfied. By remaining abstract, a family of languages and analyses are defined, which are parameterized by this logic. Any decidable temporal logic can be used to instantiate the system. In (Skalka and Smith 2004a), the logic is instantiated with a variant of the μ-calculus (Kozen 1983) known as the linear time μ-calculus (Esparza 1994).

If a check is encountered and the formula is satisfied, the reduction continues, otherwise a bad state has been reached and the computation is "stuck." Reduction may be stuck in other instances as well, such as when a non-function value is applied to an argument. Our goal in designing a type discipline is to make it impossible to assign a type to any program that might reach a stuck state during reduction. That is, the notion of stuck states is extended to include failure of temporal assertions.

1.5.2 Static approximation of trace effects

Computational traces that will arise at run-time can be conservatively approximated through a type and effect system, in which the traditional notion of a program's type is extended to take into account the traces it generates as the *effect* of the program. These effects approximate run-time traces soundly, which is to say the run-time behavior of a program must be contained within its static approximation. Thus verification of the checks in a program's abstraction implies

verification of the checks in the concrete program. The analysis is imprecise in that unrealizable traces may be included in the program's approximation, thus leading to "false alarms" where a valid program is rejected by the analysis. However, it will never be the case that a program is verified by the analysis, but fails at run-time.

A traditional type system abstracts a program into the set of values it possibly computes. A type and effect system pairs this abstraction with a description of the set of all possible effects it may cause during computation (Amtoft, Nielson, and Nielson 1999; Nielson and Nielson 1999). In the case of λ_{trace}, the effect of a program is the set of all traces that could possibly be generated at run-time. The types of programs are the same as before, but now include an effectual description of the program behavior, which is not only useful for verification, but has value for the programmer's understanding.

The effectual description of the program then becomes the input to a model checker which will perform verification. The trace effects are interpretted as a labelled transition system (LTL) generating sets of traces. Although trace effects can generate infinite sets of traces, verification is still possible (Burkart, Caucal, Moller, and Steffen 2001). "Every computation that violates a safety property has a finite prefix along which the property is violated" (Kupferman and Vardi 2001).[4] Thus verification can be performed on trace effects by considering the finite prefixes of the traces they generate.

The type system for λ_{trace} is then constructed in such a way that it enjoys type safety, whereby well-typed programs do not get stuck, implying that all checks are assured to pass at run-time. This result is proved in (Skalka and Smith 2004a).

[4]This is the definition of safety properties due to Lamport.

1.5.3 Liveness and Safety, Local and Global

There are two kinds of program properties distinguished in the literature due to Lamport: that of *liveness* and *safety* (Lamport 1977). Informally, a safety property states that "something bad will never happen." On the other hand a liveness property states that "something good will eventually happen." These notions were later formalized by Alpern and Schneider (Alpern and Schneider 1984). Such properties can hold either globally, that is for the entire behavior of the program, or locally, which is to say the property holds for some specific region of the programs execution. Such local safety properties are exemplified by access control systems such as stack inspection and history based security mechanisms (Wallach and Felten 1998; Abadi and Fournet 2003; Edjlali, Acharya, and Chaudhary 1998). It should be clear that an analysis for reasoning about local properties subsumes a global analysis simply by taking the region of interest to be the entire program. Thus, the policies focused upon in trace effect analysis are of *local* safety, that is over the temporal well-formedness of traces up to the current point in the computation for all possible executions of the program.

1.5.4 Subeffecting disciplines

With any effect analysis, there is always a need for a "subeffecting" mechanism for the sake of expressivity (Amtoft, Nielson, and Nielson 1999, pages 20–21). Without a notion of subeffecting, all computational paths must induce the exact same effects—a requirement that cripples expressivity, and allows virtually no interesting programs to be written. For instance, without a notion of weakening, two branches of an if expression will be required to induce the same effects. At first glance, this may seem not unlike requiring that both branches have the same type, certainly a reasonable requirement. However a type represents a set of values, thus requiring both branches

of an if to have the same type is to say that both branches must compute values within the same set. Imagine the stronger restriction of requiring both branches to compute the exact same value. This would be unreasonable; why should one write an if when both branches compute the same value? To require all branches of programs to compute identical values could hardly be considered a program! But this is analogous to what would happen without a subeffecting discipline. If all paths are required to have the exact same effect, then all possible executions of the program would produce this same effect. Analysis giving "for all" guarantees in such a setting would be of scant use—there is only one possible effect caused by running the program. Thus the power of our analysis stems from the points of approximation, which is precisely the purpose subeffecting serves.

The system described in this thesis relies on a subeffecting discipline of weakening. Given an expression that has some trace effect, it can be "weakened" by adding arbitrary effects, so long as the set of traces described by the original trace effect are contained within the new trace effect. When two expressions are required for the sake of typability to have the same effect, such as the branches of an if, it is then possible to weaken the effects into agreement by enlarging one of them to obtain the other.

1.5.5 The algorithm

The weakening system is presented first as a set of logical deduction rules.[5] A term has a type if there exists a derivation giving that type for the term. These logical systems provide the conceptual framework for proving properties of the analysis, such as progress and preservation, however since they are non-deterministic, they don't provide a means of mechanically constructing proofs

[5]We refrain from calling these *inference* rules so as not to be confused with the algorithmic rules for type inference.

from a term.

To reconstruct a type from a program algorithmically, a set of inference rules are given that form the basis of a type inference algorithm. By establishing the soundness of these algorithms with respect to the logical systems, it follows that properties proven of the logical system hold for the algorithms as well. In the case of the weakening analysis, we give a proof showing that a logical judgment can be constructed from a given algorithmic judgment.

The inference rules define a syntax-directed, deterministic algorithm for deducing a set of constraints on the type along with a set of constraints on the effect of a program. Such an approach is known as constraint type inference (Eifrig, Smith, and Trifonov 1995), albeit extended for the setting of type and effect inference. A program is well-typed if these constraints can be satisfied. Constraints on trace effects are given by the containment relation. Constraints on types are equality constraints, and can be solved with the well-known technique of unification which generates a substitution unifying the equations whenever one exists.[6] Constraints on effects are solved using a novel algorithm presented in (Skalka and Smith 2004a). Since effect constraints may be recursive and moreover since equality of trace effects is undecidable, unification will not suffice. The algorithm for solving effect constraint relies on an invariant on the form of constraint inequalities. All effect constraints are variable in their upper bound. Informally speaking, the least upper bound of all effects flowing into this variable are computed. Substituting this value for the variable necessarily satisfies the inequation. Simply being able to infer such terms which have undecidable identity is remarkable in its own right.

[6] According to (Pfenning 2002), the first unification algorithm was sketched as a footnote in Jacques Herbrand's Ph.D. thesis (1929), and was introduced into automated deduction by (Robinson 1971). The application to type inference was first discovered in the setting of the simply-typed lambda calculus (i.e. the monomorphic calculus) by (Hindley 1969; Milner 1978). Linear versions of the algorithm have been developed (Paterson and Wegman 1976).

The type system employs let-polymorphism. As such, polymorphic values are second class entities and can be introduced only via a "let" binding which assigns a polymorphic type to a variable in the scope of the body of the let expression. This restricted version of polymorphism is employed by STANDARD ML (Milner, Tofte, Harper, and MacQueen 1997).[7]

1.6 Outline

This thesis is organized as follows: The next section recalls and discusses the syntax and semantics of the λ_{trace} language as defined in (Skalka and Smith 2004a).

Section 3 recalls and discusses the type and effect system for trace analysis. This section includes a discussion of the background needed in the development of our results. The logical type and effect system is the presented. Having stated the logical system, the type inference system is then described. Following this the inference algorithm is proved sound with respect to the logical system, one of the novel contributions of this paper.

Section 4 describes the implementation of this type system. Section 5 provides a conclusion and offers areas for future work. Implementations of all the algorithms discussed in this thesis are included in Appendix A.

[7]The more powerful form of polymorphism which affords polymorphic values first-class status, know as polyvariants or impredicative polymorphism (or SYSTEM F), has an undecidable type inference problem (Wells 1999).

2 Core Language

This section provides the syntax and operational semantics for the language model λ_{trace}, a syntactic extension of the λ-calculus,[8] which represents the core of higher-order programming languages, and provides a concise, mathematically oriented, characterization of computation.[9] The syntactic extension includes syntactic productions for events, constants, and the language of assertions. The semantics of λ_{trace} is a modification of the contextual reduction relation for the applicative order, call-by-value fragment of the calculus. The semantics are modified to maintain a trace throughout reduction. Events and checks are recorded in the trace and checks reflect upon the trace to assert properties. The syntax and semantics are made precise below.

Using a λ-calculus setting gains leverage into analyzing arbitrary programming languages; the analysis can be scaled up by growing the language semantics or by giving a richer target language a semantics in λ_{trace}. As discussed in (Skalka, Smith, and Van Horn 2005), the analysis is language neutral in another sense in that transformations over the language of effects can be used to extend the features of the programming language in a modular fashion.

2.1 Syntax

The syntax of λ_{trace} terms is given in Figure 1. The base values include booleans and unit. The expression $\mathsf{let}\ x = v\,\mathsf{in}\,e$ binds a polymorphic value v to the identifier x in the scope of e. Recursive functions are given by the terms $\lambda_z x.e$, where x is the formal parameter of the function, and z refers to the function itself within the scope of e. Application is written $e\ e$. The following

[8] The λ-calculus was first developed by Alonzo Church to study higher-order logics, but the system turned out to be a boon to the understanding and implementation of programming languages.

[9] As contrasted with Turing machines, which are a machine-oriented characterizations of computation.

c	$\in$	$\mathcal{C}$	*atomic constants*
b	$::=$	$\mathsf{true} \mid \mathsf{false}$	*boolean values*
v	$::=$	$x \mid \lambda_z x.e \mid c \mid b \mid \neg \mid \vee \mid \wedge \mid ()$	*values*
e	$::=$	$v \mid e\, e \mid ev(e) \mid \phi(e) \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{let}\ \ x = v\, \mathsf{in}\, e$	*expressions*
η	$::=$	$\epsilon \mid ev(c) \mid \eta; \eta$	*traces*
E	$::=$	$[\,] \mid v\, E \mid E\, e \mid ev(E) \mid \phi(E) \mid \mathsf{if}\, E\, \mathsf{then}\, e\, \mathsf{else}\, e$	*evaluation contexts*

Figure 1: Language syntax

$\eta, (\lambda_z x.e)v$	$\rightsquigarrow$	$\eta, e[v/x][\lambda_z x.e/z]$		(β)
$\eta, \neg\mathsf{true}$	$\rightsquigarrow$	η, false		$(notT)$
$\eta, \neg\mathsf{false}$	$\rightsquigarrow$	η, true		$(notF)$
$\eta, \wedge\, \mathsf{true}$	$\rightsquigarrow$	$\eta, \lambda x.x$		$(andT)$
$\eta, \wedge\ \mathsf{false}$	$\rightsquigarrow$	$\eta, \lambda\_.\mathsf{false}$		$(andF)$
$\eta, \vee\, \mathsf{true}$	$\rightsquigarrow$	$\eta, \lambda\_.\mathsf{true}$		(orT)
$\eta, \vee\, \mathsf{false}$	$\rightsquigarrow$	$\eta, \lambda x.x$		(orF)
$\eta, \mathsf{if\, true\, then}\, e_1\, \mathsf{else}\, e_2$	$\rightsquigarrow$	η, e_1		(ifT)
$\eta, \mathsf{if\, false\, then}\, e_1\, \mathsf{else}\, e_2$	$\rightsquigarrow$	η, e_2		(ifF)
$\eta, \mathsf{let}\ \ x = v\, \mathsf{in}\, e$	$\rightsquigarrow$	$\eta, e[v/x]$		(let)
$\eta, ev(c)$	$\rightsquigarrow$	$\eta; ev(c), ()$		$(event)$
$\eta, \phi(c)$	$\rightsquigarrow$	$\eta; ev_\phi(c), ()$	if $\Pi(\phi(c), \hat{\eta}\ ev_\phi(c))$	$(check)$
$\eta, E[e]$	$\rightarrow$	$\eta', E[e']$	if $\eta, e \rightsquigarrow \eta', e'$	$(context)$

Figure 2: Language semantics

syntactic sugarings are assumed:

$$e_1 \wedge e_2 \triangleq \wedge e_1 e_2 \qquad e_1 \vee e_2 \triangleq \vee e_1 e_2 \qquad \lambda x.e \triangleq \lambda_z x.e \quad z \text{ not free in } e$$

$$\lambda\_.e \triangleq \lambda x.e \quad x \text{ not free in } e \qquad e_1; e_2 \triangleq (\lambda\_.e_2)(e_1)$$

Events ev are named entities parameterized by static constants c.[10] These constants $c \in \mathcal{C}$ are

[10] We occasionally omit these constants and write ev in place of $ev(c)$ when c is irrelevant to the discussion.

abstract and could be taken to be strings or IP addresses, for example, and the parameterization by constants adds to the expressive power of assertions. A trace η is an ordered sequences of these events, which maintain the sequence of events experienced during program execution. The notation $\hat{\eta}$ is used to denote the sequence obtained from η by removing the delimiters ";". Trace assertions ϕ, also named and parameterized by constants c, implement checks. The syntax of assertions is purposefully left unspecified.

2.2 Semantics

The operational semantics of λ_{trace} is defined in Figure 2 as a small step reduction relation $\rightsquigarrow$ and contextual relation $\rightarrow$ on configurations η, e, where η is the trace of run-time program events. The syntax of evaluation contexts enforces a left-to-right, call-by-value deterministic reduction strategy. The evaluation relation $\rightarrow^\star$ is the reflexive, transitive closure of the contextual reduction relation.

Note that in the *event* reduction rule, an event $ev(c)$ encountered during execution is added to the end of the trace. The *check* rule specifies that when a configuration η, ϕ is encountered during execution, the "check event" $ev_\phi(c)$ is appended to the end of η, and $\phi(c)$ is required to be satisfied by the current trace η, according to a meaning function Π, where Π is defined such that $\Pi(\phi(c), \hat{\eta})$ holds iff $\phi(c)$ is valid for $\hat{\eta}$. As noted before, the syntax and semantics of checks are left abstract for parameterization of the analysis, thus no definition of Π is given in this paper. In case a check fails at runtime, execution is "stuck"; formally:

Definition 2.1 *A configuration η, e is* stuck *iff e is not a value and there does not exist η' and e' such that $\eta, e \rightarrow \eta', e'$. If $\epsilon, e \rightarrow^\star \eta, e'$ and η, e' is stuck, then e is said to* go wrong.

The following example demonstrates the basics of syntax and operational semantics.

Example 2.1 *Let the function f be defined as:*

$$f \triangleq \lambda_z x.\mathsf{if}\, x\, \mathsf{then}\, ev_1(c)\, \mathsf{else}\, (ev_2(c); z(\mathsf{true}))$$

Then, in the operational semantics, we have:

$$\epsilon, f(\mathsf{false}) \rightarrow^\star ev_2(c); ev_1(c), ()$$

The call to f with the argument true *will cause the alternative branch of the conditional to be taken, $ev_2(c)$ is then be appended to the current trace, and a recursive call is made with the argument* true*. The consequent branch of the if will be taken, appending $ev_1(c)$ and returning unit, the final value.*

Programs can go wrong in a number of ways, such as by applying a non-function value, branching on a non-boolean value, or, most importantly for the purposes of this work, asserting a temporal property over the current trace which does not hold. Now that the operational semantics of the language have been defined, the next section defines a logical type system such that well typed programs do not go wrong. By demonstrating that type inference is sound with respect to the logic, we conclude that typed infered programs do not go wrong, and therefore our automated analysis is well-founded.

3 Weaken Analysis

In this section, we prove the soundness of a compile-time analysis algorithm conservatively approximating the traces generated by λ_{trace} programs. This result demonstrates that trace effect analysis can be effectively automated to prove temporal specifications of program events, the main constribution of this thesis.

Section 3.1 describes a logical type and effect system for λ_{trace} that statically identifies a conservative approximation of the traces that would result during execution. The approximation of this system relies on a notion of weakening as its subeffecting mechanism. Section 3.2 describes the algorithm for proof inference and in Section 3.2.3 the soundness of the system is proved with respect to the logical system. Composing the algorithmic soundness result with the logical safety result of (Skalka and Smith 2004a) yields the safety of programs well-typed by type inference.

3.1 Logical System

This section describes the logical type theory for λ_{trace} relying on weakening as its subeffecting discipline. Much of this development was given in (Skalka and Smith 2004a), which are recalled here in order to state the main results of this thesis in the following sections.

Section 3.1.1 defines the language of types and traces effects, Section 3.1.2 defines the interpretation of the trace effect language, Section 3.1.3 gives the logical typing rules, Section 3.1.4 discusses the weakening subeffecting discipline and gives examples of the expressiveness it contributes, and finally Section 3.1.7 defines trace effect validity and states the type safety property of the system.

$$
\begin{array}{lcll}
\multicolumn{4}{l}{\delta \in \mathcal{V}_s, t \in \mathcal{V}_\tau, h \in \mathcal{V}_H, \alpha, \beta \in \mathcal{V}_s \cup \mathcal{V}_\tau \cup \mathcal{V}_H} & \textit{variables} \\
s & ::= & \delta \mid c & \textit{singletons} \\
\tau & ::= & t \mid \{s\} \mid \tau \xrightarrow{H} \tau \mid \textit{bool} \mid \textit{unit} \mid s \mid H & \textit{types} \\
\sigma & ::= & \forall \bar{\alpha}.\tau & \textit{type schemes} \\
H & ::= & \epsilon \mid h \mid \textit{ev}(s) \mid H; H \mid H|H \mid \mu h.H & \textit{trace effects} \\
\Gamma & ::= & \varnothing \mid \Gamma; x : \sigma & \textit{type environments}
\end{array}
$$

Figure 3: Type syntax

3.1.1 Types and trace effects

The languages of types, trace effects, and type environments are given in Figure 3. The language of types, denoted $\mathcal{T}$, includes variables, singleton sets, functions, booleans, and unit. For technical reasons, trace effects and singletons are included in the language of types, although no program will be assigned such a type (a program may be assigned a singleton *set* type, however). A type is *basic* if it is a singleton, or the type *bool*, *unit*, or ϵ. A type is *atomic* if it is basic or a variable, otherwise the type is *composite*.

A type scheme generalizes the free variables of a type. Vector notation is used for type variables, where $\bar{\alpha}$ is taken to mean $\alpha_1 \ldots \alpha_n$ and $\forall \bar{\alpha}.\tau$ is shorthand for $\forall \alpha_1 \ldots . \forall \alpha_n . \tau$. The free variables of a type are given by the function $\mathrm{fv} : \mathcal{T} \to \mathcal{V}$, inductively defined as follows:

$$
\begin{array}{rcl}
\mathrm{fv}(\tau) & = & \varnothing \text{ if } \tau \text{ is basic} \\
\mathrm{fv}(\alpha) & = & \{\alpha\} \\
\mathrm{fv}(\{s\}) & = & \mathrm{fv}(s) \\
\mathrm{fv}(\tau \xrightarrow{H} \tau') & = & \mathrm{fv}(\tau) \cup \mathrm{fv}(H) \cup \mathrm{fv}(\tau') \\
\mathrm{fv}(\textit{ev}(s)) & = & \mathrm{fv}(s) \\
\mathrm{fv}(H; H') & = & \mathrm{fv}(H) \cup \mathrm{fv}(H') \\
\mathrm{fv}(H|H') & = & \mathrm{fv}(H) \cup \mathrm{fv}(H') \\
\mathrm{fv}(\mu h.H) & = & \mathrm{fv}(H) \setminus \{h\}
\end{array}
$$

The function is extended to type schemes, environments, and judgments respectively as follows:

$$\begin{aligned} \mathrm{fv}(\forall \bar{\alpha}.\tau) &= \mathrm{fv}(\tau) \setminus \bar{\alpha} \\ \mathrm{fv}(\varnothing) &= \varnothing \\ \mathrm{fv}(\Gamma; x : \sigma) &= \mathrm{fv}(\Gamma) \cup \mathrm{fv}(\sigma) \\ \mathrm{fv}(\Gamma, H \vdash e : \tau) &= \mathrm{fv}(\Gamma) \cup \mathrm{fv}(H) \cup \mathrm{fv}(\tau) \end{aligned}$$

A type is *closed* whenever $\mathrm{fv}(\tau) = \varnothing$ and the set of closed types is denoted $\hat{\mathcal{T}}$, known as the *ground types*.

Type environments are mappings from expression variables to type schemes and represent assumptions made about the type of variables. Environment lookup $\Gamma(x)$ is defined as follows:

$$\begin{aligned} \Gamma; x : \sigma(x) &= \sigma \\ \Gamma; y : \sigma(x) &= \Gamma(x) \end{aligned}$$

The types of Figure 3 are standard with the exception of singleton types and trace effects. Singletons are explained below and trace effects are the subject of the following section.

When viewed as a set of values, assigning a type to an expression implies that the expression evaluates to a value in that set (or diverges). In the case of a singleton set type $\{v\}$, the type is inhabited by exactly one value, namely v. Thus a singleton set type is a very precise characterization of an expression; singleton set types are used to type constants which will parameterize events and checks, adding to the expressiveness of the program logics.

3.1.2 Trace effect interpretation

Trace effects approximate traces and their syntax is very close to their trace counterparts. The interpretation of trace effects is given as a Labelled Transition System (LTS) that generates sets of traces including a distinguished $\downarrow$ symbol to denote termination.

Definition 3.1 *The interpretation of trace effects is defined via strings, possibly terminated by* $\downarrow$*, (called* traces*) denoted* θ*, over the following alphabet:*

$$\begin{array}{lcl} s & ::= & ev(c) \mid \epsilon \mid s\ s \\ a & ::= & s \mid s\downarrow \end{array}$$

We let Θ *range over prefix-closed sets of traces.*

Sets of traces are obtained by interpreting closed trace effects as programs in a simple nondeterministic transition system:

Definition 3.2 (Trace effect transition relation)

$$ev(c) \xrightarrow{ev(c)} \epsilon \qquad H_1|H_2 \xrightarrow{\epsilon} H_1 \qquad H_1|H_2 \xrightarrow{\epsilon} H_2 \qquad \mu h.H \xrightarrow{\epsilon} H[\mu h.H/h] \qquad \epsilon; H \xrightarrow{\epsilon} H$$

$$H_1; H_2 \xrightarrow{a} H_1'; H_2 \text{ if } H_1 \xrightarrow{a} H_1'$$

The sets of traces Θ associated with a closed trace effect is given by the following relation:

Definition 3.3 (Trace effect interpretation)

$$[\![H]\!] = \{a_1 \cdots a_n \mid H \xrightarrow{a_1} \cdots \xrightarrow{a_n} H'\} \cup \{a_1 \cdots a_n\downarrow \mid H \xrightarrow{a_1} \cdots \xrightarrow{a_n} \epsilon\}$$

Any trace effect interpretation is clearly prefix-closed. In this interpretation, an infinite trace is viewed as the set of its finite prefixes. Trace effect equivalence is undecidable, following from

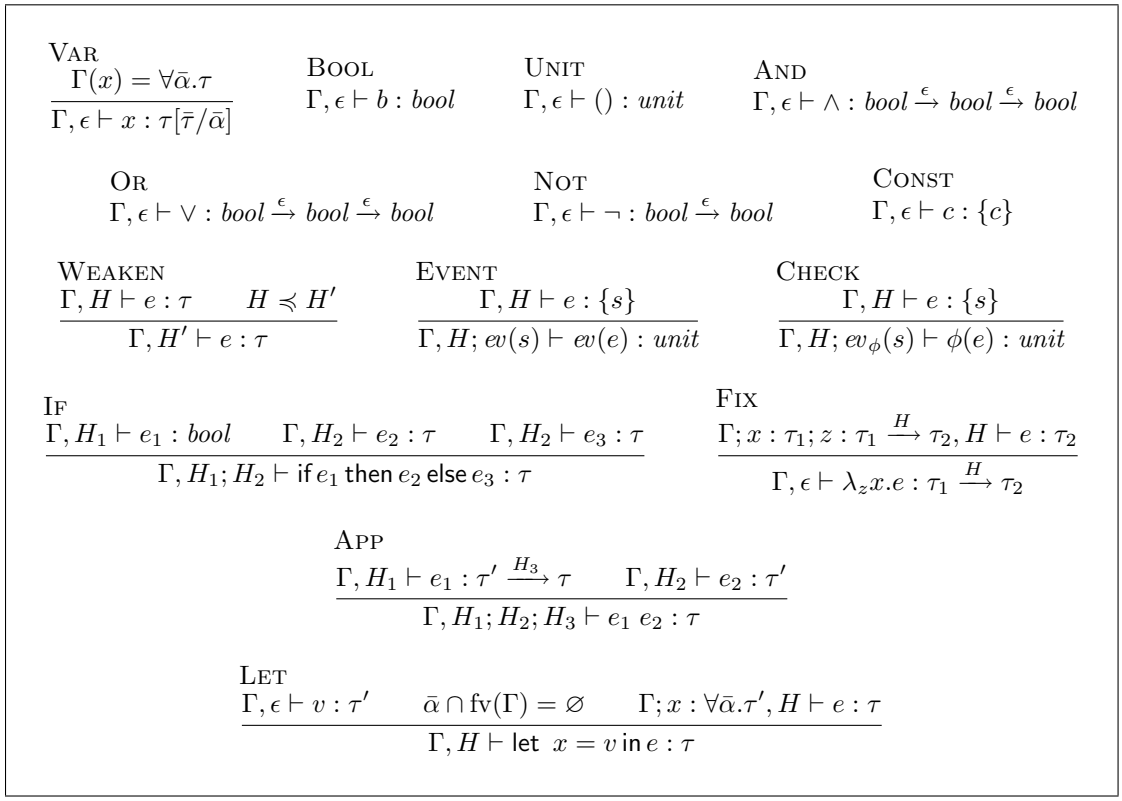

Figure 4: Logical weakening typing rules

the undecidability results for the equivalent system of basic process algebras (Burkart, Caucal, Moller, and Steffen 2001).

3.1.3 Logical judgments

Typing rules are given in Figure 4 in the form of a sequent calculus (Gentzen 1935). The rules are essentially those of a let-polymorphic Hindley-Milner (HM) system extended with trace effects and singleton types for constants. It should be noted that this type analysis is a conservative extension of HM; any event and check free program which is typable in this system is HM-typable and for any type derivable in HM, the same type is derivable in this system.

Judgments, or sequents, are of the form $\Gamma, H \vdash e : \tau$, which can be read as "under assumptions Γ, the expression e has type τ and trace effect H." We let $\mathcal{J}$ range over judgments. Type schemes, and thus judgments, are identified up to consistent renaming of bound variables.

A type judgment $\mathcal{J}$ is *derivable* iff there exists a tree of judgments having leaves which are the axioms VAR, BOOL, UNIT, AND, OR, NOT, or CONST, each node is derivable by the application of WEAKEN, EVENT, CHECK, IF, ABS, APP, or LET to the children of the node, and the tree is rooted at $\mathcal{J}$. Such a tree is a *derivation*, or in other words, $\mathcal{J}$ is *derivable*.

The VAR rule is standard; the variable's type is found in the environment and $\forall$-bound variables are instantiated. A variable has no effect, hence ϵ is given as the term's trace effect. The rules BOOL, UNIT, AND, OR, and NOT are also standard rules and have no effect. Applying the boolean functions also has no effect, hence they have ϵ as the latent effect labelled on their function types. The rule CONST is used to type constants, which parameterize events and checks. Constants have a singleton type which identifies the constant value.

The rules EVENT and CHECK assign checks and events the type *unit*. The effect of the expression is given by appending the event or check to the trace effect of the subexpression.

The IF rule requires the two branches to have the same type and effect. The expression is assigned the type of it's branches, and the effect is the concatenation of the branch term's effect and the effect of the branches.

The FIX rule assigns a function type to lambda abstractions. The environment is extended with an assumption about the lambda bound variables. The variable z is assumed to have the type of the function itself. The effect of the body of the term becomes the latent effect of the function, which will occur only when this function is applied. The function itself has no effect, hence it is assigned ϵ.

The App rule assigns a type to the application of one term to another. The applied term must have a functional type where the domain type matches the type of the applicand term. The application is assigned the effect of concatenating the effect of the applied term, the applicand term, and the latent effect which is realized when the function is applied.

The Let rule derives a type τ' for the syntactic value v and generalizes over the free variables in τ'. Notice, however, that the assumptions in Γ are never generalized, which would lead to inconsistencies.

The Weaken rule is the subject of the following section.

3.1.4 Weakening and type containment

The Weaken rule relies on a notion of *containment* on types and trace effects, that is if $\Gamma, H \vdash e : \tau$ is derivable, then $\Gamma, H' \vdash e : \tau'$ is derivable whenever H' contains H and τ' contains τ. For the purposes of the weakening system, type containment is taken simply to mean equality. Effect containment is defined in terms of traces genenerated by a trace effect. A trace effect H' contains H if the set of all traces generated by H is a subset of all traces generated by H', that is $[\![H]\!] \subseteq [\![H']\!]$, however H and H' may include trace effect variables, in which case $[\![\cdot]\!]$ is undefined. In order for the containment relation to be meaningful, containment relies on a notion of type *interpretation* which maps types to ground types. Two types are related by containment if all interpretations of the types are equal, that is, for any consistent assignment of variables to ground types, the resulting types will be syntactically identical (up to α-renaming). Likewise, H' contains H if $[\![H]\!]$ contains $[\![H']\!]$ for all consistent assigments of variables to ground trace effects in H and H'.

Definition 3.4 (Interpretation) *An* interpretation $\rho : \mathcal{V} \to \hat{\mathcal{T}}$ *is a well-kinded total mapping*

from type variables to ground types. By well-kinded it is meant that if $\alpha_i \in \mathcal{V}_s$*, then* τ_i *is of singleton kind, if* $\alpha_i \in \mathcal{V}_H$*, then* τ_i *is of trace effect kind, and so on.*

Interpretations are extended to total mappings from types to types as follows:

$$\begin{array}{rcl}
\rho(\tau) & = & \tau \text{ if } \tau \text{ is basic} \\
\rho(\tau \xrightarrow{H} \tau') & = & \rho(\tau) \xrightarrow{\rho(H)} \rho(\tau') \\
\rho(\{s\}) & = & \{\rho(s)\} \\
\rho(ev(s)) & = & ev(\rho(s)) \\
\rho(H_1; H_2) & = & \rho(H_1); \rho(H_2) \\
\rho(H_1 | H_2) & = & \rho(H_1) | \rho(H_2) \\
\rho(\mu h.H) & = & \mu h.\rho'(H) \\
 & & \quad \textit{where } \mathrm{dom}(\rho') = \mathrm{dom}(\rho) \setminus \{h\} \\
 & & \quad \textit{and } \forall \alpha \in \mathrm{dom}(\rho'), \rho(\alpha) = \rho'(\alpha)
\end{array}$$

Substitutions are extended to total mappings from type scheme to type schemes as follows:

$$\begin{array}{rcl}
\rho(\forall \alpha.\tau) & = & \forall \alpha.\rho'(\tau) \\
 & & \quad \textit{where } \mathrm{dom}(\rho') = \mathrm{dom}(\rho) \setminus \{\alpha\} \\
 & & \quad \textit{and } \forall \beta \in \mathrm{dom}(\rho'), \rho(\beta) = \rho'(\beta)
\end{array}$$

Having established the definition of interpretations, we can now formulate the containment relation, which is a partial order on trace effects:

Definition 3.5 (Type containment) $\tau \preccurlyeq \tau'$ *iff* $\rho(\tau) = \rho(\tau')$ *for all interpretations* ρ*.*

Definition 3.6 (Trace effect containment) $H \preccurlyeq H'$ *iff* $[\![\rho(H)]\!] \subseteq [\![\rho(H')]\!]$ *for all interpretations* ρ*.*

The interaction of trace effect interpretation $[\![H]\!]$ and WEAKEN is fairly subtle. Weakening allows us to replace the current trace effect with a less precise trace effect. At first glance, this

may seem unsound. Consider the following derivation where ϕ asserts that some event ev has occurred:

$$\frac{\varnothing, \epsilon \vdash c : \{c\}}{\varnothing, \epsilon; ev_\phi(c) \vdash ev(c) : unit}$$

Clearly the check should fail, since ev does not occur. However, it may seem WEAKEN can be used to approximate the effect in such a way that ev is included in the effect. That is:

$$\frac{\dfrac{\varnothing, \epsilon \vdash c : \{c\} \qquad \epsilon \preccurlyeq ev}{\varnothing, ev \vdash c : \{c\}}}{\varnothing, ev; ev_\phi(c) \vdash ev(c) : unit}$$

However, this is not a legitimate derivation because $\epsilon \not\preccurlyeq ev$. To see why this is, recall from Definition 3.3 that $[\![\epsilon]\!] = \{\downarrow\}$, $[\![ev]\!] = \{ev, ev\downarrow\}$, and $\{\downarrow\} \not\subseteq \{ev, ev\downarrow\}$. On the other hand, it is possible to weaken as follows:

$$\frac{\dfrac{\varnothing, \epsilon \vdash c : \{c\} \qquad \epsilon \preccurlyeq ev|\epsilon}{\varnothing, ev|\epsilon \vdash c : \{c\}}}{\varnothing, ev|\epsilon; ev_\phi(c) \vdash ev(c) : unit}$$

If ϕ asserts ev must have occurred, then the trace effect will not be valid since:

$$[\![ev|\epsilon; ev_\phi]\!] \quad = \quad \{ev, ev\, ev_\phi(c), ev\, ev_\phi(c)\downarrow, ev_\phi(c), ev_\phi(c)\downarrow\}$$

and ev does not occur in all of the traces. Thus the rule for weakening allows for sound approximations of trace effects.

Note that (Skalka and Smith 2004a) use the following rule:

$$\textsc{Weaken}\ \frac{\Gamma, H \vdash e : \tau}{\Gamma, H|H' \vdash e : \tau}$$

Clearly the weakening rule based on containment generalizes this notion of weakening since $H \preccurlyeq H|H'$.

It should be noted that the logical typing rules have been stated in such a way so as to rely on the containment relation to acheive a subeffecting discipline. The logical system is thus parameterized by the containment relation. The effectual weakening system is obtained by taking containment to be equality on types and trace subsumption on effects. Other subeffecting disciplines, such as subtyping, can be obtained simply by altering the containment relation. In the case of subtyping, all that would need to be changed is the type containment relation which would be given in terms of a partial order on ground types (a subtype relation), rather than equality. The typing rules need not change, however. Similarly, the type inference algorithm for a subtyping system can reuse the constraint inference rules, although other parts of the inference must be changed since unification is no longer sufficient to resolve type constraints.

3.1.5 Expressiveness of weakening

As noted in Section 1, any effect analysis needs a notion of "subeffecting." Without some notion of subeffecting, all possible paths through a computation must induce the same effects, however such a restriction is far too imposing to form the basis of a useful analysis. The system presented in this section relies on a notion of weakening for subeffecting, which allows the effect of a program to be weakened so that it may have any effect which contains the original, where containment is formally given in Definition 3.6. To see how weakening is used, consider the following program:

Example 3.1

$$\mathsf{if}\, x\, \mathsf{then}\, ev(c_1)\, \mathsf{else}\, ev(c_2)$$

This program takes a boolean value x and branches based on x's value, causing either effect $ev(c_1)$ or $ev(c_2)$ to occur. In a logical system without subeffecting, e.g. in this system without the WEAKEN rule, there would be no way to type this program since the effects of the branches are not identical as required by the IF rule. However, WEAKEN can be used to bring the effect of each branch into agreement by giving the subterm $ev(c_1)$ the less precise effect $ev(c_1)|ev(c_2)$, and likewise for $ev(c_2)$. Now the whole program has the effect $ev(c_1)|ev(c_2)$, which matches our intuition about what the program will do.

3.1.6 Expressiveness of polymorphism

Another aspect of this type system that adds to its expressiveness is parametric polymorphism. The expression $\mathsf{let}\ x = v\, \mathsf{in}\, e$ binds the variable x to the polymorphic value v in the scope of the expression e. The free type variables in the type of v are $\forall$-quantified and can take on different instantiations at each occurrence of x in e. Consider the following program:

Example 3.2

$$\mathsf{let}\ f = \lambda x.(x(); \phi)\, \mathsf{in}\, f(\lambda x.ev_1); f(\lambda x.ev_2)$$

This program binds f to a function of one argument that will first apply the argument to (), then perform a check ϕ. This function f is then applied sequentially to the argument $\lambda x.ev_1$ and $\lambda x.ev_2$. We would expect the trace of this program to be $ev_1; \phi; ev_2; \phi$. Note that operationally the semantics of $\mathsf{let}\ x = v\, \mathsf{in}\, e$ are equivalent to $(\lambda x.e)\ v$, so in a system without polymorphism, e.g. in this system without the LET rule, it would be possible to consider this

expression as syntactic sugar for $(\lambda f.f(\lambda x.ev_1); f(\lambda x.ev_2))(\lambda x.x(); \phi)$. This expression is still typable by relying on WEAKEN. The effect of ev_1 can be weakened to $ev_1|ev_2$ and likewise for ev_2, which then allows f to have the type $unit \xrightarrow{(ev_1|ev_2);\phi} unit$, and the effect of the whole program is $(ev_1|ev_2); \phi; (ev_1|ev_2); \phi$.

In a system with polymorphism, on the other hand, f can be given the type $\forall h.unit \xrightarrow{h;\phi} unit$. The two distinct applications of f can then instantiate h as ev_1 and ev_2 respectively, giving the whole program the effect $ev_1; \phi; ev_2; \phi$. This effect is a tighter characterization of the run time trace that will occur and thus may be valid in cases where the effect obtained without polymorphism would be invalid. For example, if ϕ asserts that ev_1 must have occurred, then the less precise effect will not be valid and the program will be rejected by the analysis although its clear that ev_1 must always occur before ϕ. Also note that since polymorphism gives a tighter characterization of the trace effect of the program, it not only increases the expressibility of the analysis, but combats the state explosion problem when model checking effects for validity by reducing the size of the state space.

Unlike the above example which was still typable, albeit less precisely, through the use of WEAKEN rather than polymorphism, there are other cases in which polymorphism can be used to type programs that would not otherwise be typable, The following example is such a case:

Example 3.3

$$\mathsf{let}\ \ f = \lambda x.ev(x)\ \mathsf{in}\ f(c_1); f(c_2)$$

This program binds f to a function of one argument that, when applied, triggers an event parameterized by the argument of the function. Relying on polymorphism results in the type $\forall \alpha.\alpha \xrightarrow{ev(\alpha)} unit$ for f, and each application of f can instantiate α to $\{c_1\}$ and $\{c_2\}$ respec-

tively. This program cannot be typed without polymorphism, since it would be equivalent to $(\lambda f.f(c_1); f(c_2))(\lambda x.ev(x))$, and the term $\lambda f.f(c_1); f(c_2)$ is not well-typed since it applies f to two arguments of disjoint type.

3.1.7 Validity of trace effects and Type Safety

The validity of a trace effect rests on the validity of the assertion events that occur in traces in its interpretation. In particular, for any given predicate event in a trace, that predicate must hold for the immediate prefix trace that precedes it:

Definition 3.7 *A trace effect H is* valid *iff for all $\theta ev_\phi(c) \in [\![H]\!]$ it is the case that:*

$$\Pi(\phi(c), \theta ev_\phi(c))$$

holds. A type judgment $\Gamma, H \vdash e : \tau$ is valid *iff it is derivable and H is valid.*

An important aspect of this definition is that $[\![H]\!]$ contains *prefix* traces. Essentially, if $\Gamma, H \vdash e : \tau$ is derivable, then $[\![H]\!]$ contains "snapshots" of e's potential run-time trace at every step of reduction, so that validity of H implies validity of any check that occurs at run-time, "part-way through" the full program trace as approximated by H. This is formally realized in the primary approximation result for trace effects:

Theorem 3.1 *If $\Gamma, H \vdash e : \tau$ is derivable for closed e and $\epsilon, e \rightarrow^\star \eta, e'$ then $\hat{\eta} \in [\![H]\!]$.*

which, in turn, is the basis of a type safety result for λ_{trace}:

Theorem 3.2 (Logical type safety) *If $\Gamma, H \vdash e : \tau$ is valid for closed e then e does not go wrong.*

$$
\begin{array}{rcll}
C & ::= & \mathbf{true} \mid \tau \sqsubseteq \tau \mid C \wedge C & \textit{type and effect constraints} \\
k & ::= & \tau / C & \textit{constrained types} \\
\varsigma & ::= & \forall \bar{\alpha}.k & \textit{constrained type schemes}
\end{array}
$$

Figure 5: Type and effect constraints

Proofs and details of these results are given in (Skalka and Smith 2004b).

In the next section, an algorithm for inferring a logical judgment algorithmically is defined. This is done by developing an alternative set of inference rules that are deterministic, which will comprise the inference algorithm. We show that any algorithmic judgment can be projected into the logical system in a derivability preserving way, ensuring that any algorithmic derivation has a corresponding logical derivation, and thus the inference system is sound.

3.2 Algorithmic System

In this section, an algorithm for inferring a type and effect for programs is defined and proved sound with respect to the logical system of Section 3, which is to say, this algorithm assigns a type τ and valid effect H to an expression e under assumptions Γ, if $\Gamma, H \vdash e : \tau$ is valid.

The inference algorithm is described in Section 3.2.1 which infers constrained type schemes for programs. Section 3.2.2 describes how these algorithmic judgements can be related to the logical judgements of Section 3.1. Finally, Section 3.2.3 proves that algorithmic judgments are sound with respect to the logical system.

$$\textsc{Var}\ \frac{\Gamma(x) = \forall \bar{\alpha}.k}{\Gamma, \epsilon \vdash_{\bar{\alpha}} x : k[\bar{\alpha}'/\bar{\alpha}]} \qquad \textsc{Const}\ \frac{}{\Gamma, \epsilon \vdash_{\varnothing} c : \{c\}/\mathbf{true}} \qquad \textsc{Event}\ \frac{\Gamma, H \vdash_{\mathcal{V}} e : \tau/C}{\Gamma, H; ev(\delta) \vdash_{\mathcal{V}\cup\{\delta\}} ev(e) : \mathit{unit}/C \wedge \tau \sqsubseteq \{\delta\}}$$

$$\textsc{Check}\ \frac{\Gamma, H \vdash_{\mathcal{V}} e : \tau/C}{\Gamma, H; ev_{\phi}(\delta) \vdash_{\mathcal{V}\cup\{\delta\}} \phi(e) : \mathit{unit}/C \wedge \tau \sqsubseteq \{\delta\}}$$

$$\textsc{If}\ \frac{\Gamma, H_1 \vdash_{\mathcal{V}_1} e_1 : \tau_1/C_1 \qquad \Gamma, H_2 \vdash_{\mathcal{V}_2} e_2 : \tau_2/C_2 \qquad \Gamma, H_3 \vdash_{\mathcal{V}_3} e_3 : \tau_3/C_3 \qquad \mathcal{V}_1 \sharp \mathcal{V}_2 \sharp \mathcal{V}_3}{\Gamma, H_1; H_2 | H_3 \vdash_{\mathcal{V}_1\cup\mathcal{V}_2\cup\mathcal{V}_3\cup\{t\}} \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : t/C_{1,2,3} \wedge \tau_1 \sqsubseteq \mathit{bool} \wedge \tau_2 \sqsubseteq t \wedge \tau_3 \sqsubseteq t}$$

$$\textsc{App}\ \frac{\Gamma, H_1 \vdash_{\mathcal{V}_1} e_1 : \tau_1/C_1 \qquad \Gamma, H_2 \vdash_{\mathcal{V}_2} e_2 : \tau_2/C_2 \qquad \mathcal{V}_1 \sharp \mathcal{V}_2}{\Gamma, H_1; H_2; h \vdash_{\mathcal{V}_1\cup\mathcal{V}_2\cup\{t,h\}} e_1\ e_2 : t/C_{1,2} \wedge \tau_1 \sqsubseteq \tau_2 \xrightarrow{h} t}$$

$$\textsc{Fix}\ \frac{\Gamma; x : t; z : t \xrightarrow{h} t', H \vdash_{\mathcal{V}} e : \tau/C}{\Gamma, \epsilon \vdash_{\mathcal{V}\cup\{t,t',h\}} \lambda_z x.e : t \xrightarrow{h} t'/C \wedge \tau \sqsubseteq t' \wedge H \sqsubseteq h}$$

$$\textsc{Let}\ \frac{\Gamma, \epsilon \vdash_{\mathcal{V}_1} v : \tau'/C' \qquad \Gamma; x : \forall \bar{\alpha}.\tau'/C', H \vdash_{\mathcal{V}_2} e : \tau/C \qquad \bar{\alpha} = \mathrm{fv}(\tau', C') - \mathrm{fv}(\Gamma) \qquad \mathcal{V}_1 \sharp \mathcal{V}_2}{\Gamma, H \vdash_{\mathcal{V}_1\cup\mathcal{V}_2} \mathsf{let}\ x = v\, \mathsf{in}\, e : \tau/C \wedge C'}$$

Figure 6: Algorithmic type inference rules

3.2.1 Algorithmic judgments

The language of types and constraints are given in Figure 5. Type inference rules are given in Figure 6 in a manner similar to (Eifrig, Smith, and Trifonov 1995).[11] The algorithm works as follows: inference rules are *syntax-directed*; there is a single rule which applies to each possible shape of an expression. To type an expression e, given a context Γ, the rule corresponding to the shape of e is applied, generating a constrained type scheme. Note that any syntactically well-formed program can be assigned a constrained type, however for soundness, only well-typings

[11] In Figure 6, notation is abused by writing $C_{1,2,\ldots,n}$ to mean $\bigwedge_{i=1}^{n} C_i$ so that the rules fit within the margins of this paper. This notation is not used outside of this figure.

should be considered. Intuitively, this type scheme describes the set of all possible types that could be assigned to the given expression and a program is well-typed if this set is non-empty. A precise characterization of this set of types is given later.

This presentation of type inference differs slightly from (Skalka and Smith 2004a). In particular, a conjunctive representation of constraints is employed, rather than a set representation. However, it is sometimes convenient to describe algorithms using the set representation defined by the following notation:[12]

Definition 3.8 (Constraint set notation) *Let* $\hat{C}$ *range over* atomic *constraints, i.e.:*

$$\hat{C} ::= \mathbf{true} \mid \tau \sqsubseteq \tau$$

and let $C \triangleq \bigwedge_{i=1}^{j} \hat{C}_i$, $D \triangleq \bigwedge_{i=1}^{k} \hat{D}_i$, *and* $E \triangleq \bigwedge_{i=1}^{l} \hat{E}_i$. *Then define:*

$$\begin{aligned}
\hat{C} \in C &\iff \hat{C} \in \bigcup_{i=1}^{j} \left\{\hat{C}_i\right\} \\
D \subseteq C &\iff \bigcup_{i=1}^{k} \left\{\hat{D}_i\right\} \subseteq \bigcup_{i=1}^{j} \left\{\hat{C}_i\right\} \\
C \setminus D = E &\iff \bigcup_{i=1}^{j} \left\{\hat{C}_i\right\} \setminus \bigcup_{i=1}^{k} \left\{\hat{D}_i\right\} = \bigcup_{i=1}^{l} \left\{\hat{E}_i\right\}
\end{aligned}$$

The following shorthands are used in writing algorithmic judgments:

Definition 3.9 (Algorithmic judgment shorthand notation)

$$\Gamma \vdash e : \tau \triangleq \Gamma \vdash e : \tau/\mathbf{true} \qquad\qquad \vdash e : \tau/C \triangleq \varnothing \vdash e : \tau/C$$

[12] This notation is used, for example, in the definition of the *MGS* algorithm in Figure 8.

When choosing variable names during type inference, *fresh* names should be chosen at each step. There are two notions of freshness, a name is *locally fresh* if it does not appear in any subderivation from the current point in the derivation. If a derivation is viewed as a tree with sets of variable names chosen at each node, local freshness is the property that no descendant of the current node chooses the same name as chosen by the current node. A name is *globally fresh* if it does not appear in any other part of the derivation, i.e. no descendant of any ancestor of the current node chooses the same name as chosen by the current node. Local freshness is not imposed by the typing rules, but a *canonical strategy* is assumed without loss of generality. The canonical strategy is to always choose locally fresh names. This increases the completeness of the algorithm; notice that were names chosen in a non-locally fresh way, programs would be needlessly over-constrained, reducing the completeness of the analysis, but over constraining never compromises the soundness of the algorithm.

Judgments derived under such a strategy give rise to a notion of *canonical judgments*, in which each binding in the type environment quantifies over distinct variables from all other bindings in the environment.

Definition 3.10 (Canonical judgment) *A canonical judgment is a judgment having distinct bound variables in the type environment. That is, a judgment:*

$$x_1 : \forall \bar{\alpha}_1.k_1; \ldots; x_n : \forall \bar{\alpha}_n.k_n, H \vdash_{\mathcal{W}} e : k_0$$

is canonical *iff*

$$\bigcup_{i=1}^{n} \bar{\alpha}_i \cap \bigcup_{j=0}^{n} \mathrm{fv}(k_j) \setminus \mathrm{fv}(k_i) = \varnothing$$

It should be noted that a canonical derivation strategy preserves canonical judgments.

Global freshness, on the other hand, is enforced by the typing rules. Notice that judgments explicitly pass the set of used variable names in judgments as done in (Pierce 2002). The relation $\mathcal{V}\sharp\mathcal{V}'$ holds *iff* $\mathcal{V} \cap \mathcal{V}' = \varnothing$. This makes explicit the disjunction of chosen names in each subderivation during inference. To ease the notation, the $\mathcal{V}$ annotations are frequently omitted.

Both local and global freshness of names can easily be accomplished in a implementation by using a procedure with local state, recording which names have been used, and never using those names again. In essence, the typing rules just thread this state through the derivation.

We will return to *why* these freshness conditions are needed after introducing the notions of substitutions, which are the fundamental to our inference method and constraint solutions.

3.2.2 Relating logical and algorithmic judgments

We now turn to relating the algorithmic system to the logical system of Section 3.1.

A *solved form* of an algorithmic derivation is any corresponding logical judgment. This section demonstrates that given any valid algorithmic derivation, any solved form of the derivation is a valid logical derivation. A proof is shown which gives a construction from a valid algorithmic derivation into a particular solved form. From this particular solved form, all other instances of the judgement are derivable. The notion of a derivation instance relies on *substitutions*, which will be fundamental to the inference method.

Definition 3.11 (Substitution) *A* substitution $\psi : \mathcal{V} \to \mathcal{T}$ *is a well-kinded, finite mapping from type variables to types, written* $[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ *(or* $[\bar{\tau}/\bar{\alpha}]$ *to use the vector notation shorthand), where* $\alpha_1 \ldots \alpha_n$ *are distinct type-variables, and* $\mathrm{dom}([\bar{\tau}/\bar{\alpha}]) = \bar{\alpha}$.

Substitutions are extended to total mappings from types to types as follows:

$$
\begin{array}{rcl}
\psi(\tau) & = & \tau \textit{ if } \tau \textit{ is basic} \\
\psi(\alpha) & = & \alpha \textit{ if } \alpha \notin \mathrm{dom}(\psi) \\
\psi(\tau \xrightarrow{H} \tau') & = & \psi(\tau) \xrightarrow{\psi(H)} \psi(\tau') \\
\psi(\{s\}) & = & \{\psi(s)\} \\
\psi(ev(s)) & = & ev(\psi(s)) \\
\psi(H_1; H_2) & = & \psi(H_1); \psi(H_2) \\
\psi(H_1 | H_2) & = & \psi(H_1) | \psi(H_2) \\
\psi(\mu h.H) & = & \mu h.\psi'(H) \\
 & & \textit{where } \mathrm{dom}(\psi') = \mathrm{dom}(\psi) \setminus \{h\} \\
 & & \textit{and } \forall \alpha \in \mathrm{dom}(\psi'), \psi(\alpha) = \psi'(\alpha)
\end{array}
$$

Substitutions are extended to total mappings from type scheme to type schemes as follows:

$$
\begin{array}{rcl}
\psi(\forall \alpha.\tau) & = & \forall \alpha.\psi'(\tau) \\
 & & \textit{where } \mathrm{dom}(\psi') = \mathrm{dom}(\psi) \setminus \{\alpha\} \\
 & & \textit{and } \forall \beta \in \mathrm{dom}(\psi'), \psi(\beta) = \psi'(\beta)
\end{array}
$$

Substitutions are extended to total mappings from type environments to type environments as follows:

$$
\begin{array}{rcl}
\psi(\varnothing) & = & \varnothing \\
\psi(\Gamma; x : \sigma) & = & \psi(\Gamma); x : \psi(\sigma)
\end{array}
$$

Finally, substitutions are extended to total mappings from judgments to judgments as follows:

$$
\psi(\Gamma, H \vdash e : \tau) \quad = \quad \psi(\Gamma), \psi(H) \vdash e : \psi(\tau)
$$

We say that $\psi(\tau)$ is an instance *of τ, that $\psi(\sigma)$ is an instance of σ, and so on.*

Substitutions as defined in Definition 3.11 are extended to constraints and constrained type

schemes as follows.

$$\begin{array}{rcl}
\psi(\mathbf{true}) & = & \mathbf{true} \\
\psi(\tau \sqsubseteq \tau') & = & \psi(\tau) \sqsubseteq \psi(\tau') \\
\psi(C \wedge C') & = & \psi(C) \wedge \psi(C') \\
\psi(\forall \bar{\alpha}.\tau/C) & = & \forall \bar{\alpha}.\psi'(\tau)/\psi'(C) \\
 & & \text{where } \mathrm{dom}(\psi') = \mathrm{dom}(\psi) \setminus \bar{\alpha} \\
 & & \text{and } \forall \alpha \in \mathrm{dom}(\psi'), \psi(\alpha) = \psi'(\alpha)
\end{array}$$

We turn now to the notion of a solution to a constraint where constraints are interpreted based on the containment relation, a partial ordering of ground types, as defined in Definition 3.5 and Definition 3.6.

Definition 3.12 (Solution) *A substitution ψ is a* solution *to a constraint C, written $\psi \vdash C$, iff it is derivable according to the following rules:*

$$\frac{}{\psi \vdash \mathbf{true}} \qquad \frac{\psi(\tau_1) \preccurlyeq \psi(\tau_2)}{\psi \vdash \tau_1 \sqsubseteq \tau_2} \qquad \frac{\psi \vdash C_1 \qquad \psi \vdash C_2}{\psi \vdash C_1 \wedge C_2}$$

Definition 3.13 (Constraint entailment) *We write $C \Vdash D$ iff $\psi \vdash C$ implies $\psi \vdash D$, and $C = D$ iff $C \Vdash D$ and $D \Vdash C$.*

A partial ordering on solutions in terms of their *generality* can now be defined. Later it is shown that inference always returns a solution, whenever one exists, which is most general.

Definition 3.14 (Most general solution) *If ψ and ψ' are solutions of C, the ψ is* more general *than ψ' iff there exists a substitutions ψ'' such that $\psi' = \psi'' \circ \psi$. A substitution is a* most general solution *(MGS) of C iff ψ is a solution of C and is more general than any other solution*

of C.

Definition 3.15 (Satisfiable) *A canonical derivable judgment,*

$$\mathcal{J} \quad \triangleq \quad x_1 : \forall \bar{\alpha}_1.\tau_1/C_1; \ldots; x_n : \forall \bar{\alpha}_n.\tau_n/C_n, H \vdash_{\mathcal{W}} e : \tau_0/C_0$$

is satisfiable *iff there exists ψ, such that,*

$$\psi \vdash \bigwedge_{i=0}^{n} C_i$$

in which case it is said that $\mathcal{J}$ is satisfied under ψ, written $\psi \vdash \mathcal{J}$.

Now that a notion of constraint solution and solution generality is defined, we may return to the issue of variable name choice.

To see why both local and global freshness are needed consider the typing of $\mathsf{let}\ \ f = \lambda x.x\ \mathsf{in}\ ff$, which is the identity function and the expression is expected to have type $t \to t$. The following derivation is derivable and canonical, that is, the choice of names takes a canonical strategy.[13]

Example 3.4 (Canonical derivation)

$$\dfrac{\dfrac{\overline{x : t \vdash_{\varnothing} x : t}}{\vdash_{\varnothing} \lambda x.x : t \to t} \qquad \dfrac{\overline{f : \forall t.t \to t \vdash_{\{t'\}} f : (t \to t)[t''/t]} \qquad \overline{f : \forall t.t \to t \vdash_{\{t'\}} f : (t \to t)[t'/t]}}{f : \forall t.t \to t \vdash_{\{t', t'', t'''\}} ff : t'''/t'' \to t'' \sqsubseteq (t' \to t') \to t'''}}{\vdash_{\{t', t'', t'''\}} \mathsf{let}\ \ f = \lambda x.x\ \mathsf{in}\ ff : t'''/t'' \to t'' \sqsubseteq (t' \to t') \to t'''}$$

Notice that $[t' \to t'/t'', t' \to t'/t''']$ is a most general solution of $t'' \to t'' \sqsubseteq (t' \to t') \to t'''$ and yields $t' \to t' \equiv_{\alpha} t \to t$, the expected type.[14]

[13] Since none of the functions used in this and the following examples, the derivations omit each abstraction's "self" variable and all constraints on them. They could be easily included but would only clutter the presentation.

[14] The relation $\equiv_{\alpha}$ denotes syntactic equality up to a consistent renaming of bound variables.

The following example constructs a logical derivation that is a most general solved form of the above derivation:

Example 3.5 (Most general solved form derivation)

$$\dfrac{\dfrac{\dfrac{}{x : t \vdash x : t}}{\vdash \lambda x.x : t \to t} \qquad \dfrac{\dfrac{}{f : \forall t.t \to t \vdash f : (t \to t)[t' \to t'/t]} \qquad \dfrac{}{f : \forall t.t \to t \vdash f : (t \to t)[t'/t]}}{f : \forall t.t \to t \vdash ff : t' \to t'}}{\vdash \mathsf{let}\ \ f = \lambda x.x\ \mathsf{in}\ ff : t' \to t'}$$

Notice that the most general solved form derivation of Example 3.5 is exactly the derivation obtained by applying the most general solution to each type in the algorithmic canonical derivation of Example 3.4.

Now consider if local freshness were violated by choosing a name which has already appeared in a subderivation of where the choice occurs. Suppose in the App rule another name instead of t''' were chosen, such as t', which has already appeared in the right subderivation.

Example 3.6 (Derivation violating local freshness)

$$\dfrac{\dfrac{\dfrac{}{x : t \vdash_{\varnothing} x : t}}{\vdash_{\varnothing} \lambda x.x : t \to t} \qquad \dfrac{\dfrac{}{f : \forall t.t \to t \vdash_{\{t''\}} f : t \to t[t''/t]} \qquad \dfrac{}{f : \forall t.t \to t \vdash_{\{t'\}} f : t \to t[t'/t]}}{f : \forall t.t \to t \vdash_{\{t',t''\}} ff : t'/t'' \to t'' \sqsubseteq (t' \to t') \to t'}}{\vdash_{\{t',t''\}} \mathsf{let}\ \ f = \lambda x.x\ \mathsf{in}\ ff : t'/t'' \to t'' \sqsubseteq (t' \to t') \to t'}$$

But the constraint $t'' \to t'' \sqsubseteq (t' \to t') \to t'$ has no solution. By not choosing canonically in this example, an artificial constraint $t''' = t'$ has been induced, which over-constrains the type needlessly—to the point of being unsatisfiable.[15]

Both of the previous derivations have chosen names in a globally fresh manner, which is to say the implicit conditions on $\mathcal{V} \sharp \mathcal{V}'$ hold at each step of the derivation. Suppose this requirement

[15] Interestingly, things would have worked out fine and produced a most general solved form had the non-locally fresh choice of t'' been made in the App rule. In this case, the choice of names induces a constraint already imposed by the term and is not over constraining, but rather contributing nothing.

were omitted. Returning to the above example, the name t' could have been selected instead of t''. Notice that this and all other name choices in the derivation are locally fresh. The name t' is chosen in both branches of the App rule, but in either branch t' does not appear in a subderivation when the name is chosen. Thus the App condition $t' \sharp t'$ would not hold.

Example 3.7 (Derivation violating global freshness)

$$\dfrac{\dfrac{\dfrac{}{x : t \vdash_{\varnothing} x : t}}{\vdash_{\varnothing} \lambda x.x : t \to t} \qquad \dfrac{\dfrac{}{f : \forall t.t \to t \vdash_{\{t'\}} f : t \to t[t'/t]} \qquad \dfrac{}{f : \forall t.t \to t \vdash_{\{t'\}} f : t \to t[t'/t]}}{f : \forall t.t \to t \vdash_{\{t',t'''\}} ff : t'''/t' \to t' \sqsubseteq (t' \to t') \to t'''}}{\vdash_{\{t',t'''\}} \mathsf{let}\ \ f = \lambda x.x\ \mathsf{in}\ ff : t'''/t' \to t' \sqsubseteq (t' \to t') \to t'''}$$

The constraint $t' \to t' \sqsubseteq (t' \to t') \to t'''$ does not have a solution. In fact in this example the choice of names has undone the flexibility afforded by polymorphism by unifying the separate uses of a polymorphic value. It is as though the program were instead $((\lambda f.f\ f)\ (\lambda x.x))$, which is ill-typed.

In summary, type inference chooses variable names for the result type of an application, the argument and return types of functions, the type of the branches in an if expression, and the $\forall$-quantified variables in type schemes. If names are not chosen in a fresh manner, what would otherwise be distinct types in a program are forced to be unified. This needless unification can cause programs to be algorithmically ill-typed when in fact there does exists a logical derivation for the program, and would be algorithmically well-typed under a canonical naming strategy. Particular choices of names can undo the expressiveness afforded by polymorphism by unifying the distinct occurrences of a polymorphic value. It's also possible that a non-fresh choice of names has no effect, for example if it forces the unification of types which must be unified under any naming strategy. The choice of variable names requires some special attention in the formal development, although a canonical naming strategy is easily implemented through the use of

state such as a counter that is incremented when generating variable names.

In proving soundness, we are tasked with constructing a logical judgment, given an algorithmic judgment. Looking ahead, one technical difficulty is that in a constrained type scheme, not all universally quantified type variables are polymorphic, unlike logical type schemes. Monomorphism can be indirectly imposed via monomorphic constraints. For example, the following scheme quantifies over α, but the constraint imposes monomorphism:

$$\forall\alpha.\alpha/\alpha = \beta \xrightarrow{h} \mathit{bool}$$

Such a difficulty is not novel to our system, as universally quantified variables may be monomorphically constrained in an effect–free program, which is to say this technicality arises in any constraint-based version of Milner's type inference algorithm $\mathcal{W}$ for λ-calculus extended with polymorphic-let. Although algorithm $\mathcal{W}$ has been proved sound and complete when eagerly unifying constraints, first by Milner's student Luis Damas in (Damas and Milner 1982; Damas 1985) and later as a machine checkable proof in (Naraschewski and Nipkow 1998), the correspondence between the constraint-based and eager versions of this algorithm long remained folklore without a formally established soundness and completeness result (Lee and Yi 1998). Our soundness result reproves Lee and Yi soundness result for the constraint-based $\mathcal{W}$ by virtue of the fact that the type system is a conservative extension of HM.

To see how such a constrained type can arise, consider the example of typing $\mathsf{let}\ f = \lambda x.\neg x\ \mathsf{in}\ f$. First, $\lambda x.\neg x$ is typed, obtaining the following derivation of constraint inference rules:

$$\dfrac{\dfrac{x : t \vdash \neg : \mathit{bool} \to \mathit{bool} \qquad x : t \vdash x : t}{x : t \vdash \neg x : t'/\mathit{bool} \to \mathit{bool} = t \to t'}}{\vdash \lambda x.\neg x : t \to t'/\mathit{bool} \to \mathit{bool} = t \to t'}$$

Now the body of the let expression, f, is typed. All type variables inferred in the constrained type scheme above are $\forall$-quantified, giving $\forall t, t'.t \to t'/\mathit{bool} \to \mathit{bool} = t \to t'$. Although this type appears to be polymorphic due to the $\forall$-quantifier, the constraint imposes that $t = \mathit{bool}$ and $t' = \mathit{bool}$, thus the type is equivalent to $\mathit{bool} \to \mathit{bool}/\mathbf{true}$, which is clearly monomorphic.

A notion of *variance* is used to distinguish between polymorphic and monomorphic quantified type variables. Any quantified type variable which is monomorphically constrained is said to be *invariant*.

Definition 3.16 (Variant) *The variable $\alpha \in \bar{\alpha}$ is* variant *in $\bar{\alpha}$ for C iff* $\mathrm{fv}(\tau) \cap \bar{\alpha} = \varnothing$ *implies* $\alpha \notin \mathrm{fv}(\psi(\tau))$ *for all most general solutions of C.*

The notion of variance is more intuitive in light of the following characterization:

Proposition 3.1 *If α is invariant in $\bar{\alpha}$ for $C \wedge C[\bar{\beta}/\bar{\alpha}]$, then $C \wedge C[\bar{\beta}/\bar{\alpha}] \Vdash \alpha \sqsubseteq \alpha[\bar{\beta}/\bar{\alpha}]$ when* $\bar{\beta} \sharp \mathrm{fv}(C)$.

The proof of this proposition appears later in Lemma 3.6 after the needed auxiliary lemmas have been developed.

Revisiting our example, $\forall t, t'.t \to t'/\mathit{bool} \to \mathit{bool} = t \to t'$, t, t' are invariant in t, t' for $\mathit{bool} \to \mathit{bool} = t \to t'$. Suppose ψ solves $\mathit{bool} \to \mathit{bool} = t \to t' \wedge \mathit{bool} \to \mathit{bool} = s \to s'$. By Definition 3.12, $\psi(\mathit{bool} \to \mathit{bool}) = \psi(t \to t')$ and $\psi(\mathit{bool} \to \mathit{bool}) = \psi(s \to s')$, which implies $\psi(t) = \psi(s)$ and $\psi(t') = \psi(s')$.

On the other hand, it should be clear that quantified variables in a polymorphic type are variant. Consider the expression $\mathsf{let}\ f = \lambda x.x\ \mathsf{in}\ f$. First, the subexpression $\lambda x.x$ is typed, obtaining the following derivation:

$$\frac{x : t \vdash x : t}{\vdash \lambda x.x : t \to t}$$

The free variables are then $\forall$-quantified giving $\forall t.t \to t/\mathbf{true}$ and t is clearly variant in **true**– there is no constraint to possibly impose monomorphism.

This example can be modified by η-expansion of the identity function in order to get a non-trivial constraint, that is, consider typing $\mathsf{let}\ f = \lambda x.(\lambda y.y)\ x\ \mathsf{in}\ f$. The following derivation is obtained when typing $\lambda x.(\lambda y.y)\ x$:

$$\frac{\dfrac{\dfrac{x : t; y : t' \vdash y : t'}{x : t \vdash \lambda y.y : t' \to t'} \qquad \dfrac{}{x : t \vdash x : t}}{x : t \vdash (\lambda y.y)\ x : t'' / t' \to t' = t \to t''}}{\vdash \lambda x.(\lambda y.y)\ x : t \to t'' / t' \to t' = t \to t''}$$

The free variables are $\forall$-quantified, giving $\forall t, t', t''.t \to t''/t' \to t' = t \to t''$. Suppose ψ solves $t' \to t' = t \to t'' \wedge s' \to s' = s \to s''$. By definition, $\psi(t' \to t') = \psi(t \to t'')$ and $\psi(s' \to s') = \psi(s \to s'')$, but it does not follow that $\psi(t) = \psi(s), \psi(t') = \psi(s')$, or $\psi(t'') = \psi(s'')$, thus t is truly polymorphic.

We can now give a precise characterization of the set of types denoted by a constrained type scheme. This set is determined by the set of substitutions which solve the accrued constraint. For the constrained type scheme $\forall \bar{\alpha}.\tau/C$, suppose $\psi \vdash C$, then it denotes the set of all instances of $\forall \bar{\alpha}'.\psi(\tau)$ where $\bar{\alpha}'$ is variant in $\bar{\alpha}$ for C. If there is no such ψ, then this set is empty; the expression is ill-typed. Thus type inference is reduced to deciding if C has a solution. For

constructing $\psi \vdash C$, the unification algorithm and the MGS_H algorithm are used, given in Figure 7 and Figure 8, respectively.

A notion of solved form can be defined now that definitions of solutions and variant type variables are in place. A solved form is the logical avitar of an algorithmic judgment. The soundness proof constructs a solved form from the given algorithmic judgment and demonstrates it's derivability.

Definition 3.17 (Solved form) *Given a derivable judgment, satisfied under ψ,*

$$\mathcal{J} \triangleq x_1 : \forall\bar{\alpha}_1.\tau_1/C_1; \ldots; x_n : \forall\bar{\alpha}_n.\tau_n/C_n, H \vdash_{\mathcal{W}} e : \tau_0/C_0$$

The logical judgment,

$$\mathcal{J}' \triangleq x_1 : \forall\bar{\alpha}'_1.\psi(\tau_1); \ldots; x_n : \forall\bar{\alpha}'_n.\psi(\tau_n), \psi(H) \vdash e : \psi(\tau_0)$$

where all $\bar{\alpha}'_i$ are the variants in their respective $\bar{\alpha}_i$ for $\psi \vdash \bigwedge_{i=0}^{n} C_i$, is a solved form *of $\mathcal{J}$ under ψ.*

Now we can prove soundness in a manner allowing the induction to go through. In particular, the result is parameterized by a "global" constraint C_G, which can be specialized to the top-level constraints for the top-level inference judgments.

3.2.3 Soundness

In this section, our main technical contributions are stated and proved. In particular the algorithmic type safety result is demonstrated:

$$
\begin{array}{rcl}
\mathit{unify}(\mathbf{true}) & = & \varnothing \\
\mathit{unify}(C \wedge \tau = \tau) & = & \mathit{unify}(C) \\
\mathit{unify}(C \wedge \beta = \tau) & = & \textbf{fail} \text{ if } \beta \in \mathrm{fv}(\tau), \text{ else} \\
 & & \mathit{unify}(C[\tau/\beta]) \circ [\tau/\beta] \\
\mathit{unify}(C \wedge \tau = \beta) & = & \textbf{fail} \text{ if } \beta \in \mathrm{fv}(\tau), \text{ else} \\
 & & \mathit{unify}(C[\tau/\beta]) \circ [\tau/\beta] \\
\mathit{unify}(C \wedge \{s_1\} = \{s_2\}) & = & \mathit{unify}(C \wedge s_1 = s_2) \\
\mathit{unify}(C \wedge \tau_1 \xrightarrow{H} \tau_2 = \tau_1' \xrightarrow{H'} \tau_2') & = & \mathit{unify}(C \wedge H = H' \wedge \tau_1 = \tau_1' \wedge \tau_2 = \tau_2')
\end{array}
$$

Figure 7: Constraint set unification algorithm

$$
\begin{array}{rcl}
\mathit{MGS}(C) & = & \text{let } \psi_1 = \mathit{unify}(C \setminus C') \text{ in } \mathit{MGS}_{\mathbf{H}}(\psi_1(C')) \circ \psi_1 \\
 & & \text{where } C' = \{H \sqsubseteq H' \mid H \sqsubseteq H' \in C\} \\
 & & \\
\mathit{bounds}(h, C) & = & H_1 | \cdots | H_n \text{ where } \{H_1, \ldots, H_n\} = \{H \mid H \sqsubseteq h \in C\} \\
 & & \\
\mathit{MGS}_{\mathbf{H}}(\varnothing) & = & \varnothing \\
\mathit{MGS}_{\mathbf{H}}(C) & = & \text{let } \psi = [h' | \mu h.\mathit{bounds}(h, C)/h] \text{ in} \\
 & & \mathit{MGS}_{\mathbf{H}}(\psi(C \setminus \{H \sqsubseteq h \mid H \sqsubseteq h \in C\})) \circ \psi \\
 & & \text{where } h' \text{ fresh}
\end{array}
$$

Figure 8: Most general solution algorithm

Theorem 3.3 (Algorithmic type safety) *If* $\Gamma, H \vdash_{\mathcal{W}} e : \tau/C$ *is valid for closed* e*, then* e *does not go wrong.*

Proof. Immediate from Theorem 3.4 and Theorem 3.2. □

This result is obtained by composing this the logical type safety result found in (Skalka and Smith 2004a) and the following theorem:

Theorem 3.4 (Soundness of inference) *If* $\varnothing, H \vdash_{\mathcal{W}} e : \tau/C$ *is satisfiable, then* $\varnothing, \psi(H) \vdash$

$e : \psi(\tau)$ *is derivable where* $\psi = MGS(C)$.

First, several lemmas needed to obtain the above result are proved.

Lemma 3.1 *If* ψ *is a most general solution of* C*, then* $C \Vdash \tau \sqsubseteq \psi(\tau)$.

Proof. For this result, it is sufficient to demonstrate that for any ψ' which is a solution of C, it is the case that $\psi'(\tau) \preccurlyeq \psi'(\psi(\tau))$. By definition of most general solution, Definition 3.14, there exists ψ'' such that $\psi' = \psi'' \circ \psi$. Substituting equals for equals obtains $\psi''(\psi(\tau)) \preccurlyeq \psi''(\psi(\psi(\tau)))$, and by idempotency of solutions, $\psi''(\psi(\tau)) \preccurlyeq \psi''(\psi(\tau))$ which is true by reflexivity of containment. □

Lemma 3.2 $\psi \circ [\bar{\beta}/\bar{\alpha}] \vdash C$ *iff* $\psi \vdash C[\bar{\beta}/\bar{\alpha}]$.

Proof. $\psi \circ [\bar{\beta}/\bar{\alpha}] \vdash C$ iff $(\psi \circ [\bar{\beta}/\bar{\alpha}])(\tau_1) \sqsubseteq (\psi \circ [\bar{\beta}/\bar{\alpha}])(\tau_2)$ for all $\tau_1 \sqsubseteq \tau_2 \in C$, which is equivalent to $\psi(\tau_1[\bar{\beta}/\bar{\alpha}]) \sqsubseteq \psi(\tau_2[\bar{\beta}/\bar{\alpha}])$ and holds iff $\psi(\tau_1') \sqsubseteq \psi(\tau_2')$ for all $\tau_1' \sqsubseteq \tau_2' \in C[\bar{\beta}/\bar{\alpha}]$. □

Lemma 3.3 *If* $\psi \vdash C \wedge C[\bar{\beta}/\bar{\alpha}]$ *then* $\psi \circ [\bar{\beta}/\bar{\alpha}] \vdash C \wedge C[\bar{\beta}/\bar{\alpha}]$.

Proof. First observe the following equivalence by Lemma 3.2:

$$\psi \vdash C[\bar{\beta}/\bar{\alpha}] \iff \psi \circ [\bar{\beta}/\bar{\alpha}] \vdash C$$

What remains is to show $\psi \circ [\bar{\beta}/\bar{\alpha}] \vdash C[\bar{\beta}/\bar{\alpha}]$. By assumption, $\psi \vdash C[\bar{\beta}/\bar{\alpha}]$. Observe that $C[\bar{\beta}/\bar{\alpha}] = C[\bar{\beta}/\bar{\alpha}][\bar{\beta}/\bar{\alpha}]$, thus $\psi \vdash C[\bar{\beta}/\bar{\alpha}][\bar{\beta}/\bar{\alpha}]$ and by Lemma 3.2, $\psi \circ [\bar{\beta}/\bar{\alpha}] \vdash C[\bar{\beta}/\bar{\alpha}]$. □

Note that this lemma does not apply in the other direction as evinced by the following counter example:

$$\begin{array}{rcl} \psi & \triangleq & [\gamma'/\alpha] \circ [\gamma/\beta] \\ C & \triangleq & \alpha \sqsubseteq \gamma \end{array}$$

Thus $\psi \circ [\beta/\alpha] \vdash C \wedge C[\beta/\alpha]$, but $\psi \nvdash C \wedge C[\beta/\alpha]$ since $\gamma' \not\sqsubseteq \gamma$.

Lemma 3.4 *If* $C \wedge C[\bar{\beta}/\bar{\alpha}] \Vdash \tau_1 \sqsubseteq \tau_2$ *then* $C \wedge C[\bar{\beta}/\bar{\alpha}] \Vdash \tau_1[\bar{\beta}/\bar{\alpha}] \sqsubseteq \tau_2[\bar{\beta}/\bar{\alpha}]$.

Proof. Suppose ψ is a solution of $C \wedge C[\bar{\beta}/\bar{\alpha}]$, then $\psi \circ [\bar{\beta}/\bar{\alpha}]$ is also a solution by Lemma 3.3, and $\psi \circ [\bar{\beta}/\bar{\alpha}] \vdash \tau_1 \sqsubseteq \tau_2$ by Definition 3.13. It follows by Lemma 3.2 that $\psi \vdash \tau_1[\bar{\beta}/\bar{\alpha}] \sqsubseteq \tau_2[\bar{\beta}/\bar{\alpha}]$. □

The implication can be proved in the opposite direction with the restriction $\bar{\beta} \sharp \mathrm{fv}(C, \tau_1, \tau_2)$.

Lemma 3.5 *If* $C \wedge C[\bar{\beta}/\bar{\alpha}] \Vdash \tau_1[\bar{\beta}/\bar{\alpha}] \sqsubseteq \tau_2[\bar{\beta}/\bar{\alpha}]$ *then* $C \wedge C[\bar{\beta}/\bar{\alpha}] \Vdash \tau_1 \sqsubseteq \tau_2$ *when* $\bar{\beta} \sharp \mathrm{fv}(C, \tau_1, \tau_2)$.

Proof. Suppose ψ is a solution of $C \wedge C[\bar{\beta}/\bar{\alpha}]$. By assumption, $\psi \vdash C$, and since $\bar{\beta} \sharp \mathrm{fv}(C)$, then $C[\bar{\alpha}/\bar{\beta}] = C$. Thus, $\psi \circ [\bar{\alpha}/\bar{\beta}] \vdash C$, but is also a solution of $C[\bar{\beta}/\bar{\alpha}]$, since $\psi \vdash C$ implies $\psi \vdash C[\bar{\beta}/\bar{\alpha}][\bar{\alpha}/\bar{\beta}]$ when $\bar{\beta} \sharp \mathrm{fv}(C)$, so $\psi \circ [\bar{\alpha}/\bar{\beta}] \vdash C[\bar{\beta}/\bar{\alpha}]$ by Lemma 3.2. Therefore $\psi \circ [\bar{\beta}/\bar{\alpha}] \vdash \tau_1[\bar{\beta}/\bar{\alpha}] \sqsubseteq \tau_2[\bar{\beta}/\bar{\alpha}]$ by Definition 3.13. By Lemma 3.2, $\psi \vdash \tau_1[\bar{\beta}/\bar{\alpha}][\bar{\beta}/\bar{\alpha}] \sqsubseteq \tau_2[\bar{\beta}/\bar{\alpha}][\bar{\beta}/\bar{\alpha}]$, which is to say $\psi \vdash \tau_1 \sqsubseteq \tau_2$ since $\bar{\beta} \sharp \mathrm{fv}(\tau_1, \tau_2)$. □

Notice that the condition $\bar{\beta} \notin \mathrm{fv}(\tau_1, \tau_2)$ is necessary since this lemma doesn't hold otherwise. Let $\tau_1 \triangleq \alpha$ and $\tau_2 \triangleq \beta$, thus $\tau_1[\beta/\alpha] \sqsubseteq \tau_2[\beta/\alpha] = \beta \sqsubseteq \beta$. Any C gives $C \Vdash \beta = \beta$, so any one such that $\beta \notin \mathrm{fv}(C)$ may be chosen. Let $C \triangleq \mathbf{true}$. Any substitution solves C, including the empty substitution, but then $\nvdash \alpha \sqsubseteq \beta$.

Corollary 3.1 $C \wedge C[\bar{\beta}/\bar{\alpha}] \Vdash \tau_1 \sqsubseteq \tau_2$ *iff* $C \wedge C[\bar{\beta}/\bar{\alpha}] \Vdash \tau_1[\bar{\beta}/\bar{\alpha}] \sqsubseteq \tau_2[\bar{\beta}/\bar{\alpha}]$ *and* $\bar{\beta} \notin \mathrm{fv}(C, \tau_1, \tau_2)$.

Proof. Immediate in the left to right direction from Lemma 3.4, and in the right to left direction from Lemma 3.5. □

We can now prove the earlier Proposition 3.1.

Lemma 3.6 *If α is invariant in $\bar{\alpha}$ for $C \wedge C[\bar{\beta}/\bar{\alpha}]$, then $C \wedge C[\bar{\beta}/\bar{\alpha}] \Vdash \alpha \sqsubseteq \alpha[\bar{\beta}/\bar{\alpha}]$ when $\bar{\beta} \sharp \mathrm{fv}(C)$.*

Proof. By assumption and Definition 3.16, there exists $\alpha' \notin \bar{\alpha}$ and most general solution ψ of $C \wedge [\bar{\beta}/\bar{\alpha}]$ such that $\alpha \in \psi(\bar{\alpha}')$. But $C \wedge C[\bar{\beta}/\bar{\alpha}] \Vdash \bar{\alpha}' \sqsubseteq \psi(\bar{\alpha}')$ by Lemma 3.1, so $C \wedge C[\bar{\beta}/\bar{\alpha}] \Vdash \bar{\alpha}' \sqsubseteq \psi(\alpha')[\bar{\beta}/\bar{\alpha}]$ by Lemma 3.4. Thus, $C \wedge C[\bar{\beta}/\bar{\alpha}] \Vdash \psi(\alpha') \sqsubseteq \psi(\alpha')[\bar{\beta}/\bar{\alpha}]$ and since $\alpha \in \psi(\alpha')$ by assumption, the result follows in a straightforward manner by induction on $\psi(\alpha')$. □

Lemma 3.7 *Let $\bar{\beta} \sharp \mathrm{fv}(C)$ and $\bar{\alpha}'$ be invariant in $\bar{\alpha}$ for $C \wedge C[\bar{\beta}/\bar{\alpha}]$ and $[\bar{\beta}/\bar{\alpha}] = [\bar{\beta}'/\bar{\alpha}', \dots]$, then for all τ and most general solutions ψ of $C \wedge C[\bar{\beta}/\bar{\alpha}]$ it is the case that $(\psi(\tau))[\psi(\bar{\beta}')/\bar{\alpha}'] \sqsubseteq \psi(\tau[\bar{\beta}/\bar{\alpha}])$.*

Proof. By Lemma 3.1, $C \wedge C[\bar{\beta}/\bar{\alpha}] \Vdash \tau \sqsubseteq \psi(\tau)$, so $C \wedge C[\bar{\beta}/\bar{\alpha}] \Vdash \tau[\bar{\beta}/\bar{\alpha}] \sqsubseteq (\psi(\tau))[\bar{\beta}/\bar{\alpha}]$ by Lemma 3.4. This and Definition 3.13 implies:

$$\psi(\tau[\bar{\beta}/\bar{\alpha}]) \quad = \quad \psi((\psi(\tau))[\bar{\beta}/\bar{\alpha}])$$

By Lemma 3.6 and Definition 3.13:

$$\psi((\psi(\tau))[\bar{\beta}/\bar{\alpha}]) \quad = \quad \psi((\psi(\tau))[\bar{\beta}/\bar{\alpha}])$$

Clearly $\psi((\psi(\tau))[\bar{\beta}'/\bar{\alpha}']) = (\psi(\tau)[\psi(\bar{\beta}')/\bar{\alpha}']$. □

The *MGS* algorithm is used to construct the most general solution of a constraint when one exists. Although a constraint can in general be of the form $H = H'$ or $H \sqsubseteq H'$, both of which are undecidable, the algorithm exploits two invariants of inference generated constraints that allow them to be solved programmatically. The first is that atomic constraints between non-effect types are *variable in effect*. So for example, constraints between function types will always be of the form:

$$\tau_1 \xrightarrow{h_1} \tau_1' \sqsubseteq \tau_2 \xrightarrow{h_2} \tau_2'$$

Where the constraints $\tau_1 \sqsubseteq \tau_2$ and $\tau_2' \sqsubseteq \tau_1'$ will likewise be variable in effect. Such a constraint can be solved through unification since one variable can be substituted for the other.

Definition 3.18 (Variable in effect) *A type τ, which is not a trace effect, is* variable in effect *iff H is a strict subterm of τ, implies $H \in \mathcal{V}_H$. This notion is extended to type environments and constraints; an environment Γ is variable in effect iff it maps every expression variable in its domain to a type which is variable in effect; a constraint C is variable in effect iff all inequalities in C are between types which are variable in effect.*

Lemma 3.8 (Variable in effect invariant of inference) *If $\Gamma, H \vdash_{\mathcal{W}} e : \tau/C$ is derivable and Γ is variable in effect, then τ/C is variable in effect.*

Proof. By induction on the derivation of $\Gamma, H \vdash_{\mathcal{W}} e : \tau/C$, reasoning by case analysis on the last step used in the derivation.

- Var.

  This case follows since Γ is variable in effect.

- Const, Event, Check, If, and App.

  These case follow in a similar manner. In each case the antecedent(s) do not extend the type environment, so the induction hypothesis applies. Then notice that the conjoined constraints in the consequent are variable in effect, so the cases hold.

- Fix.

In the antecedent, the environment is extended with bindings which are variable in effect, so the induction hypothesis applies, and the case holds since $t \xrightarrow{h} \tau/\tau \sqsubseteq t' \wedge H \sqsubseteq h$ is variable in effect iff τ is, and τ is by the inductive hypothesis.

- LET.

  The environment is extended with $x : \forall\bar{\alpha}.\tau'/C'$ in the FIX case, but τ'/C' is variable in effect by the induction hypothesis, so $\tau/C \wedge C'$ is variable in effect, which proves this case.

□

Lemma 3.9 (Soundness of unify) *For any variable in effect constraint C, unify(C) returns a solution of C if one exists, and fails otherwise.*

Proof. By straightforward extension of, for example, page 328 of (Pierce 2002).

The second invariant on constraints crucial to the decidability of type inference is that atomic constraints between effects form a system of lower bounds on effect variables. This is to say that all effect constraints are of the form $H \sqsubseteq h$. Constraints in such a form are easily solvable by joining all effects flowing in to a particular variable. For example, the following subsitution solves a constraint of this form:

$$[(\mu h.H_1|H_2|\ldots|H_n)/h] \vdash H_1 \sqsubseteq h \wedge H_2 \sqsubseteq h \wedge \ldots \wedge H_n \sqsubseteq h$$

Since:

$$H_i \preccurlyeq \mu h.H_1|H_2|\ldots|H_n$$

Note that the μ is needed since h may appear free in H_i. We refer to this invariant as "friendliness".

Definition 3.19 (Friendly) *A constraint C is* friendly *iff all atomic constraints between trace effects are of the form $H \sqsubseteq h$. This notion is extended to environments; an environment is friendly when it maps each variable to some τ/C where C is friendly.*

Lemma 3.10 (Friendly invariant of inference) *If $\Gamma, H \vdash_{\mathcal{W}} e : \tau/C$ and Γ is friendly, then C is friendly.*

Proof. By induction on the derivation of $\Gamma, H \vdash_{\mathcal{W}} e : \tau/C$. In the VAR case, C is friendly since $C = C'[\bar{\beta}/\bar{\alpha}]$ for some C' in Γ.

The only rule which extends Γ with a constraint type is LET. But then C' is friendly by the inductive hypothesis, so $\Gamma; \forall\bar{\alpha}.\tau'/C'$ is friendly, and the inductive hypothesis applies, giving that $C \wedge C'$ is friendly. The rest of the cases follow straightforwardly by observing that in each case the conjoined constraints in the consequent are friendly. □

We now show that MGS_H returns a most general solution. (The following two lemmas are stated without proof, which remains for future work).

Lemma 3.11 (Correctness of $MGS_{\mathbf{H}}$) *For any friendly effect constraint C, $MGS_H(C)$ is a most general solution of C.*

Lemma 3.12 (Correctness of MGS) *For any friendly C, $MGS(C)$ is a most general solution of C.*

We assume the idempotency of solutions. This is done without loss of generality. Observe that any solution can be given as an idempotent solution: MGS returns a solution if one exists by Lemma 3.12, and it is idempotent.

The following lemma is used to reconstruct the form of subderivations given a derivable judgment by inversion of the typing relation.

Lemma 3.13 (Inversion of inference relation)

- *if* $\Gamma, H \vdash_{\mathcal{V}} x : \tau/C$, *then* $H = \epsilon$, $\Gamma(x) = \forall \bar{\alpha}.k$, $\mathcal{V} = \bar{\alpha}$ *and there exists* $[\bar{\beta}/\bar{\alpha}]$ *such that* $\tau/C = k[\bar{\beta}/\bar{\alpha}]$ *and* $\bar{\beta} \notin \mathrm{fv}(k)$.

- *if* $\Gamma, H \vdash_{\mathcal{V}} c : \tau/C$, *then* $H = \epsilon$, $\tau = \{c\}$, $C = \mathbf{true}$, *and* $\mathcal{V} = \varnothing$.

- *if* $\Gamma, H \vdash_{\mathcal{V}} ev(e) : \tau/C$, *then* $\tau = unit$ *and there exists a judgment* $\Gamma, H' \vdash_{\mathcal{V}'} e : \tau'/C'$ *such that* $H = H'; ev(\delta)$, $C = C' \wedge \tau' \sqsubseteq \{\delta\}$, *and* $\mathcal{V}' = \mathcal{V} \cup \{\delta\}$.

- *if* $\Gamma, H \vdash_{\mathcal{V}} \phi(e) : \tau/C$, *then* $\tau = unit$ *and there exists a judgment* $\Gamma, H' \vdash_{\mathcal{V}'} e : \tau'/C'$ *such that* $H = H'; ev_{\phi}(\delta)$, $C = C' \wedge \tau' \sqsubseteq \{\delta\}$, *and* $\mathcal{V}' = \mathcal{V} \cup \{\delta\}$.

- *if* $\Gamma, H \vdash_{\mathcal{V}} \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 : \tau/C$, *then* $\tau = t$ *and there exists judgments,* $\Gamma, H_1 \vdash_{\mathcal{V}_1} e_1 : \tau_1/C_1$, $\Gamma, H_2 \vdash_{\mathcal{V}_2} e_2 : \tau_2/C_2$, *and* $\Gamma, H_3 \vdash_{\mathcal{V}_3} e_3 : \tau_3/C_3$ *such that* $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2 \cup \mathcal{V}_3 \cup \{t\}$, $\mathcal{V}_1 \sharp \mathcal{V}_2 \sharp \mathcal{V}_3$, $H = H_1; H_2 | H_3$, *and* $C = C_1 \wedge C_2 \wedge C_3 \wedge \tau_1 \sqsubseteq bool \wedge \tau_2 \sqsubseteq t \wedge \tau_3 \sqsubseteq t$.

- *if* $\Gamma, H \vdash_{\mathcal{V}} e_1\ e_2 : \tau/C$, *then* $\tau = t$ *and there exists judgments,* $\Gamma, H_1 \vdash_{\mathcal{V}_1} e_1 : \tau_1/C_1$ *and* $\Gamma, H_2 \vdash_{\mathcal{V}_2} e_2 : \tau_2/C_2$ *such that* $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2 \cup \{t, h\}$, $\mathcal{V}_1 \sharp \mathcal{V}_2$, $H = H_1; H_2; h$, *and* $C = C_1 \wedge C_2 \wedge \tau_1 \sqsubseteq \tau_2 \xrightarrow{h} t$.

- *if* $\Gamma, H \vdash_{\mathcal{V}} \lambda_z x.e : \tau/C$, *then* $\tau = t \xrightarrow{h} t'$, $H = \epsilon$, *and there exists a judgment* $\Gamma; x : t; z : t \xrightarrow{h} t', H' \vdash_{\mathcal{V}'} e : \tau'/C'$ *such that* $C = C' \wedge \tau' \sqsubseteq t' \wedge H' \sqsubseteq h$, $\mathcal{V} = \mathcal{V}' \cup \{t, t', h\}$.

- *if* $\Gamma, H \vdash_{\mathcal{V}} \mathsf{let}\ x = v \,\mathsf{in}\, e : \tau/C$, *then there exists judgments,* $\Gamma, \epsilon \vdash_{\mathcal{V}_1} v : \tau'/C'$ *and* $\Gamma; \forall \bar{\alpha}.\tau'/C', H \vdash_{\mathcal{V}_2} e : \tau/C''$ *and there exists* $[\bar{\alpha}'/\bar{\alpha}]$ *such that* $[\bar{\alpha}'/\bar{\alpha}]C' \wedge C'' = C$, $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$, *and* $\mathcal{V}_1 \sharp \mathcal{V}_2$.

Proof. Immediate from definition of the type inference relation given in Figure 6. □

We now prove the following main lemma:

Lemma 3.14 *If* $\Gamma, H \vdash_{\mathcal{W}} e : \tau/C$ *is derivable, then so is any most general solved form of* $\Gamma, H \vdash_{\mathcal{W}} e : \tau/C \wedge C_G$*, where* C_G *is arbitrary.*

Proof. By induction on the derivation of $\mathcal{J} \triangleq \Gamma, H \vdash_{\mathcal{W}} e : \tau/C$, reasoning by case analysis on the last rule used in the derivation. In each case, a logical judgment is constructed such that it is a most general solved form of $\Gamma, H \vdash_{\mathcal{W}} e : \tau/C \wedge C_G$ under ψ and then is shown to be logically derivable.

- Case: VAR, $e = x$

  Let:

$$\begin{array}{rcl} \Gamma & \triangleq & x_1 : \forall \bar{\alpha}_1.\tau_1/C_1; \ldots; x_n : \forall \bar{\alpha}_n.\tau_n/C_n \\ C' & \triangleq & C \wedge C_1 \wedge \ldots \wedge C_n \wedge C_G \end{array}$$

  By Lemma 3.13, $H = \epsilon$, $\tau/C = (\tau_i/C_i)[\bar{\beta}/\bar{\alpha}_i]$ and the last step in the derivation of $\mathcal{J}$ is:

$$\frac{\Gamma(x) = \forall \bar{\alpha}_i.\tau_i/C_i}{\Gamma, \epsilon \vdash_{\bar{\alpha}_i} x : (\tau_i/C_i)[\bar{\beta}/\bar{\alpha}_i]}$$

  where $\bar{\beta}$ fresh and $i \in \{1..n\}$. Clearly there exists C'' such that:

$$C' = C'' \wedge C_i \wedge C_i[\bar{\beta}/\bar{\alpha}_i]$$

  Since $\Gamma, H \vdash_{\mathcal{W}} e : \tau/C \wedge C_G$ has a solved form, it is canonical, hence $\bar{\alpha}_i \cap \mathrm{fv}(C'') = \varnothing$, implying $C''[\bar{\beta}/\bar{\alpha}_i] = C''$ and:

$$C' = (C'' \wedge C_i) \wedge (C'' \wedge C_i)[\bar{\beta}/\bar{\alpha}]$$

Furthermore, letting:

$$\Gamma' \quad \triangleq \quad x_i : \forall \bar{\alpha}'_1.\psi(\tau_1); \ldots ; x_n : \forall \bar{\alpha}'_n.\psi(\tau_n)$$

where $\bar{\alpha}'_i$ are variant in $\bar{\alpha}_i$ for C' for $i \in \{1..n\}$, we have that $\Gamma', H \vdash e : \psi(\tau)$ is a solved form of $\Gamma, H \vdash_{\mathcal{W}} e : \tau/C \wedge C_G$. The following judgment can be derived by the logical rule VAR:

$$\frac{\Gamma'(x) = \forall \bar{\alpha}'_i.\psi(\tau_i)}{\Gamma', \epsilon \vdash x : (\psi(\tau_i))[\psi(\bar{\beta}')/\bar{\alpha}'_i]}$$

where $[\bar{\beta}/\bar{\alpha}_i] = [\bar{\beta}'/\bar{\alpha}'_i, \ldots]$, and since:

$$\psi(\tau) \quad = \quad \psi(\tau_i[\bar{\beta}/\bar{\alpha}_i])$$

By Lemma 3.7:

$$\psi(\tau) \quad = \quad (\psi(\tau_i))[\psi(\bar{\beta}')/\bar{\alpha}'_i])$$

so this case holds.

- Case: CONST, $e = c$

  By Lemma 3.13, $H = \epsilon$ and $\tau/C = \{c\}/\mathbf{true}$. By assumption, C_G has a most general solution ψ, so $\Gamma', \psi(\epsilon) \vdash_{\mathcal{V}} c : \psi\{c\}$ is a most general solved form of $\Gamma, \epsilon \vdash_{\mathcal{V}} c : \{c\}/C_G$ and

is derivable by the logical rule CONST since $\psi(\epsilon) = \epsilon$ and $\psi(\{c\}) = \{c\}$.

- Case: EVENT, $e = ev(e')$

  By Lemma 3.13, $\tau = \mathit{unit}$ and there exists a judgment

$$\mathcal{J}_1 = \Gamma, H' \vdash e' : \tau'/C'$$

  where:

$$\begin{array}{rcl} H & = & H'; ev(\delta) \\ C & = & C' \wedge \tau' \sqsubseteq \{\delta\} \end{array}$$

  $C_G \wedge C' \wedge \tau' \sqsubseteq \{\delta\}$ has a solution, so the inductive hypothesis applies to the judgment $\Gamma, H' \vdash e' : \tau'/C_G \wedge C' \wedge \tau' \sqsubseteq \{\delta\}$, which therefore has a derivable most general solved form under ψ, namely $\Gamma', \psi(H') \vdash e' : \psi(\tau')$. Note that $\psi(\tau') = \psi(\{\delta\})$ by the fact that ψ is a solution of $\tau' \sqsubseteq \{\delta\}$ and the definition of $\sqsubseteq$. But $\Gamma', \psi(H'); ev(\psi(\delta)) \vdash ev(e') : \mathit{unit}$ is derivable by the logical rule EVENT, and is a most general solved form of $\Gamma, H \vdash e : \tau$ since

$$\begin{array}{rcl} \psi(H'); ev(\psi(\tau')) & = & \psi(H'); \psi(ev(\delta)) \\ & = & \psi(H'; ev(\delta)) \\ & = & \psi(H) \\ \mathit{unit} & = & \psi(\mathit{unit}) \\ & = & \psi(\tau) \end{array}$$

  So the case holds.

- Case: CHECK, $e = \phi(e')$

The result follows by a similar argument to the EVENT case.

- Case: IF, $e = \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3$

  By Lemma 3.13, $\tau = t$, and there exists judgments:

$$\mathcal{J}_i \triangleq \Gamma, H_i \vdash_{\mathcal{W}} e_i : \tau_i / C_i \qquad i \in \{1..3\}$$

  where:

$$\begin{array}{rcl} H & = & H_1; (H_2 | H_3) \\ C & = & C_1 \wedge C_2 \wedge C_3 \wedge \tau_1 \sqsubseteq \mathit{bool} \wedge \tau_2 \sqsubseteq t \wedge \tau_3 \sqsubseteq t \end{array}$$

  $C_G \wedge C_1 \wedge C_2 \wedge C_3 \wedge \tau_1 \sqsubseteq \mathit{bool} \wedge \tau_2 \sqsubseteq t \wedge \tau_3 \sqsubseteq t$ has a solution, so the inductive hypothesis applies to the judgments $\Gamma, H_i \vdash e_i : \tau_i / C_G \wedge C_1 \wedge C_2 \wedge C_3 \wedge \tau_1 \sqsubseteq \mathit{bool} \wedge \tau_2 \sqsubseteq t \wedge \tau_3 \sqsubseteq t$, which therefore have derivable most general solved forms under ψ, namely $\Gamma', \psi(H_i) \vdash e_i : \psi(\tau_i)$. Note that we are free to choose the same most general solution in each case since the constraint is the same. The logical rule WEAKEN can be used to construct the following derivations:

$$\mathcal{J}_{w1} \triangleq \frac{\Gamma', \psi(H_2) \vdash e_2 : \psi(t) \qquad \psi(H_2) \preccurlyeq \psi(H_2) | \psi(H_3)}{\Gamma', \psi(H_2) | \psi(H_3) \vdash e_2 : \psi(t)}$$

$$\mathcal{J}_{w2} \triangleq \frac{\Gamma', \psi(H_3) \vdash e_3 : \psi(t) \qquad \psi(H_3) \preccurlyeq \psi(H_2) | \psi(H_3)}{\Gamma', \psi(H_2) | \psi(H_3) \vdash e_3 : \psi(t)}$$

  Now that the effect of the branches are in agreement, the following derivation can be constructed using the logical rule IF:

$$\frac{\Gamma', \psi(H_1) \vdash e_1 : \mathit{bool} \qquad \mathcal{J}_{w1} \qquad \mathcal{J}_{w2}}{\Gamma', \psi(H_1); (\psi(H_2)|\psi(H_3)) \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : \psi(t)}$$

Which shows a most general solved form of $\Gamma, H \vdash e : \tau$ is derivable since:

$$\begin{array}{rcl} \psi(H_1); (\psi(H_2)|\psi(H_3)) & = & \psi(H_1; (H_2|H_3)) \\ & = & \psi(H) \\ \psi(t) & = & \psi(\tau) \end{array}$$

So the case holds.

- Case: App, $e = e_1\ e_2$

  By Lemma 3.13, $\tau = t$, and there exists judgments:

$$\mathcal{J}_i \triangleq \Gamma, H_i \vdash_{\mathcal{W}} e_i : \tau_i / C_i \qquad i \in \{1..2\}$$

  where:

$$\begin{array}{rcl} H & = & H_1; H_2; h \\ C & = & C_1 \wedge C_2 \wedge \tau_1 \sqsubseteq \tau_2 \xrightarrow{h} t \end{array}$$

  $C_G \wedge C_1 \wedge C_2 \wedge \tau_1 \sqsubseteq \tau_2 \xrightarrow{h} t$ has a solution, so the inductive hypothesis applies to the judgments $\Gamma, H_i \vdash e_i : \tau_i / C_G \wedge C_1 \wedge C_2 \wedge \tau_1 \sqsubseteq \tau_2 \xrightarrow{h} t$, which therefore have derivable most general solved forms under ψ, namely $\Gamma', \psi(H_i) \vdash e_i : \psi(\tau_i)$.

  The following derivation can be constructed using App:

$$\frac{\Gamma', \psi(H_1) \vdash e_1 : \psi(\tau_2) \xrightarrow{\psi(h)} \psi(t) \qquad \Gamma', \psi(H_2) \vdash e_2 : \psi(\tau_2)}{\Gamma', \psi(H_1); \psi(H_2); \psi(h) \vdash e_1\ e_2 : \psi(t)}$$

Which shows a most general solved form of $\Gamma, H \vdash e : \tau$ is derivable since:

$$\begin{array}{rcl} \psi(H_1); \psi(H_2); \psi(h) & = & \psi(H_1; H_2; h) \\ & = & \psi(H) \\ \psi(t) & = & \psi(\tau) \end{array}$$

So the case holds.

- Case: FIX, $e = \lambda_z x.e'$

  By Lemma 3.13, $\tau = t \xrightarrow{h} t'$, $H = \epsilon$, and there exists a judgment:

$$\mathcal{J}_1 \quad \triangleq \quad \Gamma; x : t; z : t \xrightarrow{h} t', H' \vdash_{\mathcal{W}} e' : \tau'/C'$$

  Where:

$$C \quad = \quad C' \wedge \tau' \sqsubseteq t' \wedge H' \sqsubseteq h$$

  $C_G \wedge C' \wedge \tau' \sqsubseteq t' \wedge H' \sqsubseteq h$ has a solution, so the inductive hypothesis applies to the judgment $\Gamma; x : t; z : t \xrightarrow{h} t', H' \vdash e' : \tau'/C_G \wedge C' \wedge \tau' \sqsubseteq t' \wedge H' \sqsubseteq h$, which therefore has a derivable most general solved form under ψ, namely $\Gamma'; x : \psi(t); z : \psi(t \xrightarrow{h} t'), \psi(H') \vdash e' : \psi(\tau')$.

  Note that $\psi(t \xrightarrow{h} t') = \psi(t) \xrightarrow{\psi(h)} \psi(t')$. Therefore, the following derivation can be constructed using the logical rules WEAKEN and FIX:

$$\dfrac{\dfrac{\Gamma'; x : \psi(t); z : \psi(t) \xrightarrow{\psi(h)} \psi(t'), \psi(H') \vdash e' : \psi(t') \qquad \psi(H') \preccurlyeq \psi(h)}{\Gamma'; x : \psi(t); z : \psi(t) \xrightarrow{\psi(h)} \psi(t'), \psi(h) \vdash e' : \psi(t')}}{\Gamma', \epsilon \vdash \lambda_z x.e' : \psi(t) \xrightarrow{\psi(h)} \psi(t')}$$

Which shows a most general solved form of $\Gamma, H \vdash e : \tau$ is derivable since:

$$\begin{array}{rcl} \psi(t) \xrightarrow{\psi(h)} \psi(t') & = & \psi(t \xrightarrow{h} t') \\ & = & \psi(\tau) \\ \epsilon & = & \psi(\epsilon) \\ & = & \psi(H) \end{array}$$

So the case holds.

- Case: LET, $e = \mathsf{let}\ \ x = v\ \mathsf{in}\ e'$

  By Lemma 3.13, there exists judgments:

$$\begin{array}{rcl} \mathcal{J}_1 & = & \Gamma, \epsilon \vdash v : \tau'/C' \\ \mathcal{J}_2 & = & \Gamma; x : \forall \bar{\alpha}.\tau'/C', H \vdash e' : \tau/C'' \end{array}$$

  Where:

$$\begin{array}{rcl} \bar{\alpha} & = & \mathrm{fv}(\tau', C') - \mathrm{fv}(\Gamma) \\ C & = & C' \wedge C'' \end{array}$$

  $C_G \wedge C' \wedge C''$ has a solution, so the inductive hypothesis applies to the judgment $\Gamma, H' \vdash$

$v : \tau'/C_G \wedge C' \wedge C''$, which therefore has a derivable most general solved form under ψ, namely $\Gamma', \epsilon \vdash v : \psi(\tau')$. Let $\bar{\alpha}'$ be those variables in $\bar{\alpha}$ which are variant for $C' \wedge C'' \wedge C_G$. Then $\Gamma'; \forall\bar{\alpha}'.\psi(\tau'), \psi(H) \vdash e' : \psi(\tau')$ is a most general solved form of $\Gamma; x : \forall\bar{\alpha}'.\tau'/C', H \vdash e' : \tau/C' \wedge C'' \wedge C_G$, and so is also derivable by the induction hypothesis. By assumption, $\bar{\alpha}\sharp\mathrm{fv}(\Gamma)$, which implies $\bar{\alpha}\sharp\mathrm{fv}(\Gamma')$ by Definition 3.16, so the following is derivable by the logical rule LET:

$$\frac{\Gamma', \psi(\epsilon) \vdash v : \psi(\tau') \qquad \bar{\alpha}'\sharp\mathrm{fv}(\Gamma') \qquad \Gamma'; x : \forall\bar{\alpha}'.\psi(\tau'), \psi(H) \vdash e' : \psi(\tau)}{\Gamma', \psi(H) \vdash \mathsf{let}\ \ x = v\,\mathsf{in}\,e' : \psi(\tau)}$$

So this case holds.

□

$$
\begin{array}{lcl}
MGS^*_{\mathbf{H}}(\varnothing) & = & \varnothing \\
MGS^*_{\mathbf{H}}(C) & = & \text{let } \psi = [\mu h.bounds(h, C)/h] \text{ in} \\
 & & MGS^*_{\mathbf{H}}(\psi(C \setminus \{H \sqsubseteq h \in C\})) \circ \psi
\end{array}
$$

Figure 9: Non-Most general solution algorithm

3.3 Digressions

This section includes a number of digressions from the main theme of this thesis. Section 3.3.1 discusses the *MGS* algorithm and how the algorithm presented in this thesis is a correction of the algorithm appearing in (Skalka and Smith 2004a) that does not in fact return a most general solution. Section 3.3.3 and Section 3.3.2 examine two simple mechanisms for infering more compact representations of types and constraints. The first relies on an number of auxiliary inference rules that are special or derived cases of the general rules given in Figure 6. By distinguishing these cases, certain type and effect information can be safely discarded resulting in a more effecient inference algorithm and smaller input to the model checking algorithm. The other mechanism employs a trace effect transformation algorithm that simplifies effects in a meaning preserving way in order to reduce the input to the model checker. Both mechanisms have been implemented and are included in Appendix A.

3.3.1 The MGS algorithm

The MGS algorithm given in Figure 8 is a corrected version of the algorithm appearing in (Skalka and Smith 2004a). The original presentation of the MGS algorithm, referred to here as MGS^*, is given in Figure 9. The distinguishing feature is that the correct algorithm always inserts a fresh variable into the system of lower bounds on h, whereas MGS^* does not. Without remaining

abstract in the lower bounds, the MGS^* algorithm returns a solution which is not most general, as the following example demonstrates.

Example 3.8

$$\lambda f.\mathsf{if}\ \mathbf{true}\ \mathsf{then}\ \lambda x.ev_1\ \mathsf{else}\ f$$

This function takes a function f as a parameter, branches on a boolean value, and returns either $\lambda x.ev_1$ or the given function f. The argument f is forced to be a function by virtue of its type being unified with $\lambda x.ev_1$, but as will be shown, the only requirement that should be made on the effect of f is that it contains ev_1. However, the MGS^* algorithm imposes the requirement that the effect of f be *exactly* ev_1. The algorithmic derivation for this term is the following:

$$\begin{array}{rcl} C_1 & \triangleq & t_1' \sqsubseteq \mathit{unit} \wedge h \sqsubseteq ev_1 \\ \mathcal{J}_1 & \triangleq & \dfrac{f : t, x : t_1, ev_1 \vdash_{\varnothing} ev_1 : \mathit{unit}/\mathbf{true}}{f : t, \epsilon \vdash_{\{t_1, t_1', h\}} \lambda x.ev_1 : t_1 \xrightarrow{h} t_1'/C_1} \\ C_2 & \triangleq & C_1 \wedge t_2/t_1 \xrightarrow{h} t_1' \sqsubseteq t_2 \wedge t \sqsubseteq t_2 \end{array}$$

$$\dfrac{\dfrac{f : t, \epsilon \vdash_{\varnothing} \mathbf{true} : \mathit{bool}/\mathbf{true} \qquad \mathcal{J}_1 \qquad f : t, \epsilon \vdash_{\varnothing} f : t/\mathbf{true}}{f : t, \epsilon \vdash_{\{t_1, t_1', t_2, h\}} \mathsf{if}\ \mathbf{true}\ \mathsf{then}\ \lambda x.ev_1\ \mathsf{else}\ f : t_2/C_2}}{\varnothing, \epsilon \vdash_{\{t, t', t_1, t_1', t_2, h, h'\}} \lambda f.\mathsf{if}\ \mathbf{true}\ \mathsf{then}\ \lambda x.ev_1\ \mathsf{else}\ f : t \xrightarrow{h'} t'/C_2 \wedge t_2 \sqsubseteq t' \wedge \epsilon \sqsubseteq h'}$$

Solving this constraint with MGS^* gives the following type for the program:

$$(t \xrightarrow{ev_1} \mathit{unit}) \xrightarrow{\epsilon} (t \xrightarrow{ev_1} \mathit{unit})$$

Observe that this is not a principal type by examining the logical derivation of this term where $\Gamma \triangleq x : t \xrightarrow{ev_1} unit$:

$$\dfrac{\dfrac{}{\Gamma, \epsilon \vdash \mathbf{true} : bool} \quad \dfrac{\Gamma; x : t, ev_1 \vdash ev_1 : unit}{\Gamma, \epsilon \vdash \lambda x.ev_1 : t \xrightarrow{ev_1} unit} \quad \dfrac{\Gamma(f) = t \xrightarrow{ev_1} unit}{\Gamma, \epsilon \vdash f : t \xrightarrow{ev_1} unit}}{\dfrac{\Gamma, \epsilon \vdash \mathsf{if}\,\mathbf{true}\,\mathsf{then}\,\lambda x.ev_1\,\mathsf{else}\,f : t \xrightarrow{ev_1} unit}{\varnothing, \epsilon \vdash \lambda f.\mathsf{if}\,\mathbf{true}\,\mathsf{then}\,\lambda x.ev_1\,\mathsf{else}\,f : (t \xrightarrow{ev_1} unit) \xrightarrow{\epsilon} (t \xrightarrow{ev_1} unit)}}$$

However, the following logical derivation is also derivable for this term where $\Gamma' \triangleq t \xrightarrow{ev_1|ev_2} unit$:

$$\dfrac{\dfrac{}{\Gamma', \epsilon \vdash \mathbf{true} : bool} \quad \dfrac{\dfrac{\Gamma'; x : t, ev_1 \vdash ev_1 : unit}{\Gamma'; x : t, ev_1|ev_2 \vdash ev_1 : unit}}{\Gamma', \epsilon \vdash \lambda x.ev_1 : t \xrightarrow{ev_1|ev_2} unit} \quad \dfrac{\Gamma'(f) = t \xrightarrow{ev_1|ev_2} unit}{\Gamma', \epsilon \vdash f : t \xrightarrow{ev_1|ev_2} unit}}{\dfrac{\Gamma', \epsilon \vdash \mathsf{if}\,\mathbf{true}\,\mathsf{then}\,\lambda x.ev_1\,\mathsf{else}\,f : t \xrightarrow{ev_1|ev_2} unit}{\varnothing, \epsilon \vdash \lambda f.\mathsf{if}\,\mathbf{true}\,\mathsf{then}\,\lambda x.ev_1\,\mathsf{else}\,f : (t \xrightarrow{ev_1|ev_2} unit) \xrightarrow{\epsilon} (t \xrightarrow{ev_1|ev_2} unit)}}$$

This shows that the type obtained from MGS^* is not a principal type since there is no substitution mapping type 1 to 2 below:

$$(t \xrightarrow{ev_1} unit) \quad \xrightarrow{\epsilon} \quad (t \xrightarrow{ev_1} unit) \tag{1}$$

$$(t \xrightarrow{ev_1|ev_2} unit) \quad \xrightarrow{\epsilon} \quad (t \xrightarrow{ev_1|ev_2} unit) \tag{2}$$

On the other hand, the corrected algorithm presented in Figure 8 gives the following type:

$$(t \xrightarrow{ev_1|h} unit) \xrightarrow{\epsilon|h'} (t \xrightarrow{ev_1|h} unit)$$

This is a principal type for this function. Observe that both of the above less general instances can be obtained from the substitutions $[\epsilon/h', ev_1/h]$ and $[\epsilon/h', ev_2/h]$, respectively.

The idea of continually remaining partially abstract on the lower bounds for h is related to the prior work on records in higher order languages such as the projective lambda calculus (Rémy 1992). A function with the type $t \xrightarrow{ev_1} unit$ might be viewed as a kind of record type that identifies all records consisting solely of the field ev_1. On the other hand, $t \xrightarrow{ev_1|h} unit$ identifies records consisting of *at least* the field ev_1.

A practical consequence of the MGS^* algorithm is a loss of modularity. For example, the following type might be given by separate compilation:

$$\vdash \lambda f.\mathsf{if}\, b\, \mathsf{then}\, \lambda x.ev_1\, \mathsf{else}\, f : (t \xrightarrow{ev_1} unit) \xrightarrow{\epsilon} (t \xrightarrow{ev_1} unit)$$

But then the term $(\lambda f.\mathsf{if}\, b\, \mathsf{then}\, \lambda x.ev_1\, \mathsf{else}\, f)\ \lambda x.ev_2$ cannot be analyzed in a modular fashion; the type must be re-inferred over the whole program. When a most general type is given by the analysis, there is no need to re-examine the function being applied.

3.3.2 Simplification

This section describes a trace effect transformation used to simplify effects to obtain more compact representations of the program's trace. The trace effects that arise as a result of inference

$$\epsilon; H \rightarrow_{simp} H \qquad H; \epsilon \rightarrow_{simp} H \qquad \frac{H_1 \rightarrow_{simp} H_1'}{H_1; H_2 \rightarrow_{simp} H_1'; H_2} \qquad \frac{H_2 \rightarrow_{simp} H_2'}{H_1; H_2 \rightarrow_{simp} H_1; H_2'}$$

$$H|H \rightarrow_{simp} H \qquad \frac{H_1 \rightarrow_{simp} H_1'}{H_1|H_2 \rightarrow_{simp} H_1'|H_2} \qquad \frac{H_2 \rightarrow_{simp} H_2'}{H_1|H_2 \rightarrow_{simp} H_1|H_2'} \qquad \frac{h \notin \mathrm{fv}(H)}{\mu h.H \rightarrow_{simp} H}$$

Figure 10: Effect simplification rewrite rules

are often times large and unwieldy. This section describes an algorithm for reducing trace effects to more compact descriptions.

There is no algorithm that when given a trace effect can return a unique and minimal trace effect describing the same set of effects. However, some set of reductions can be performed that will frequently produce much more understandable trace effects. The trace effect reduction rules are given in Figure 10.

The following example shows terms with their type obtained from running the inference algorithm, then computing and applying the most general solution as given by MGS, together with a simplified type where latent effects have been reduced according to the above rules:

Example 3.9 (Inference result and simplification)

- $\lambda f.f\ (\lambda x.ev_i)$

$$((t_3 \xrightarrow{(\mu h_2.(\epsilon;ev_i)|h_4)} unit) \xrightarrow{h_3} t_2) \xrightarrow{(\mu h_1.(\epsilon;(\epsilon;h_3))|h_5)} t_2$$

$$((t_3 \xrightarrow{(ev_i|h_4)} unit) \xrightarrow{h_3} t_2) \xrightarrow{(h_3|h_5)} t_2$$

- $\lambda x.\lambda f.\mathsf{if}\, x\, \mathsf{then}\, (f\ (\lambda x.ev_i))\, \mathsf{else}\, (f\ (\lambda x.ev_j))$

$$bool \xrightarrow{(\mu h_6.\epsilon|h_{12})} ((t_{13} \xrightarrow{(\mu h_8.((\epsilon;ev_i)|(\epsilon;ev_j))|h_{13})} unit) \xrightarrow{h_9} t_9) \xrightarrow{(\mu h_7.(\epsilon;((\epsilon;(\epsilon;h_9))|(\epsilon;(\epsilon;h_9))))|h_{14})} t_9$$

$$bool \xrightarrow{\epsilon|h_{12}} ((t_{13} \xrightarrow{(ev_i|ev_j)|h_{13}} unit) \xrightarrow{h_9} t_9) \xrightarrow{h_9|h_{14}} t_9$$

- $\lambda f.ev_{p:acct};(\lambda x.ev_{p:acct};ev_{enable:r:filew}(x);(f\ x))$

$$(\alpha \xrightarrow{h_{17}} t) \xrightarrow{(\mu h_{15}.((\epsilon;ev_{p:acct});\epsilon)|h_{18})} \alpha \xrightarrow{(\mu h_{16}.((\epsilon;ev_{p:acct});((\epsilon;ev_{enable:r:filew}(\alpha));(\epsilon;(\epsilon;h_{17}))))|h_{19})} t$$

$$(\alpha \xrightarrow{h_{17}} t) \xrightarrow{(ev_{p:acct}|h_{18})} \alpha \xrightarrow{((ev_{p:acct};(ev_{enable:r:filew}(\alpha);h_{17}))|h_{19})} t$$

3.3.3 Direct inference rules

In this section, alternative inference rules are considered that maintain less information during inference while remaining sound and complete with respect to the original algorithm.

We might consider a direct typing rule for sequences of expression $e_1; e_2$, such as:

$$\frac{\Gamma, H_1 \vdash e_1 : \tau' \qquad \Gamma, H_2 \vdash e_2 : \tau}{\Gamma, H_1; H_2 \vdash e_1;\ e_2 : \tau}\ \textsc{Seq}$$

It is possible to show this rule is superfluous. Suppose there exists a derivation of $\Gamma, H_1; H_2 \vdash e_1;\ e_2 : \tau$ using the Seq rule above. The expression $e_1; e_2$ is syntactic sugar for $(\lambda x.e_2)\ e_1$, where $x \notin \mathrm{fv}(e_2)$. By inversion on Seq, there exists judgments $\Gamma, H_1 \vdash e_1 : \tau'$ and $\Gamma, H_2 \vdash e_2 : \tau$. But note the following equivalence when $x, z \notin \mathrm{fv}(e_2)$ with τ' and H arbitrary.

$$\Gamma, H_2 \vdash e_2 : \tau \quad \Longleftrightarrow \quad \Gamma; x : \tau'; z : \tau' \xrightarrow{H} \tau, H_2 \vdash e_2 : \tau$$

Thus the following is derivable:

$$\frac{\dfrac{\Gamma; x : \tau'; z : \tau' \xrightarrow{H} \tau, H_2 \vdash e_2 : \tau}{\Gamma, \epsilon \vdash \lambda_z x.e_2 : \tau' \xrightarrow{H_2} \tau} \qquad \dfrac{}{\Gamma, H_1 \vdash e_1 : \tau'}}{\Gamma, H_1; H_2 \vdash (\lambda_z x.e_2)\ e_1 : \tau}$$

Since any derivation that relies on the Seq rule can be expanded into a derivation that does not, we have added no expressive power to the logical system. However, having a direct inference rule for Seq leads to more compact constrained types. Consider the following direct inference rule based on the logical Seq rule above:

$$\text{Seq}\ \frac{\Gamma, H_1 \vdash_{\mathcal{V}_1} e_1 : \tau_1/C_1 \qquad \Gamma, H_2 \vdash_{\mathcal{V}_2} e_2 : \tau_2/C_2 \qquad \mathcal{V}_1 \sharp \mathcal{V}_2}{\Gamma, H_1; H_2 \vdash_{\mathcal{V}_1 \cup \mathcal{V}_2} e_1;\ e_2 : \tau_2/C_1 \wedge C_2}$$

Without this rule, occurrences of the expression $e_1; e_2$ would be inferred as follows:

$$\frac{\dfrac{\Gamma; x : t; z : t \xrightarrow{h} t', H_2 \vdash_{\mathcal{V}_2} e_2 : \tau_2/C_2}{\Gamma, \epsilon \vdash_{\mathcal{V}_2 \cup \{t,t',h\}} \lambda_z x.e_2 : t \xrightarrow{h} t'/C_2 \wedge \tau_2 \sqsubseteq t' \wedge H_2 \sqsubseteq h} \qquad \dfrac{}{\Gamma, H_1 \vdash_{\mathcal{V}_1} e_1 : \tau_1/C_1}}{\Gamma, H_1; h' \vdash_{\mathcal{V}_1 \cup \mathcal{V}_2 \cup \{t_2,t,t',h,h'\}} (\lambda_z x.e_2)\ e_1 : t_2/C_1 \wedge C_2 \wedge \tau_2 \sqsubseteq t' \wedge H_2 \sqsubseteq h \wedge (t \xrightarrow{h} t') \sqsubseteq (\tau_1 \xrightarrow{h'} t_2)}$$

Likewise, we might consider a direct rule for non-recursive functions:

$$\textsc{Fun}\ \frac{\Gamma; x : \tau', H \vdash e : \tau}{\Gamma, \epsilon \vdash \lambda x.e : \tau' \xrightarrow{H} \tau}$$

With the corresponding inference rule:

$$\textsc{Fun}\ \frac{\Gamma; x : t, H \vdash_{\mathcal{V}} e : \tau / C}{\Gamma, \epsilon \vdash_{\mathcal{V} \cup \{t, t', h\}} \lambda x.e : t \xrightarrow{h} t' / C \wedge \tau \sqsubseteq t' \wedge h \sqsubseteq H}$$

Both of these direct rules improve the amount of information the algorithm needs to keep track of during inference and eliminates needs constraints from the resulting type. Unlike the simplification transformation, these reductions can be performed *before* other transformations such as exceptionization and stackification, which in turn reduces the time and space needed to perform these transformation, and the resulting trace effect is more compact.

4 Implementation

Implementations of all the algorithms described in this document are given in the appendix. Moreover, a full implementation of the λ_{trace} language has be developed and is available to perform trace effect analysis for λ_{trace} programs.

In addition to the algorithms presented in this thesis, the trace effect transformations presented in (Skalka, Smith, and Van Horn 2005) have been implemented and are included in Appendix A. These effect post-processing techniques include the exnization and stackification transformation. Stackification returns the stack context generated by a program given its trace effect. The stack context is useful in enforcing stack-based security mechanisms, such as Java's stack inspection mechanism (Wallach and Felten 1998). Exnization allows the language to be extended with an exception mechanism without need to redesign the inference algorithm.

4.1 Overview

The implementation is written in the OCaml programming language (Leroy 2004) and is structured using a slightly modified version of the D Development Kit (DDK) accompanying Scott Smith's textbook, *Programming Languages* (Smith 2002).

The source code included in Appendix A implements the algorithms discussed in this paper, but is a subset of the complete source code needed to run the system. In particular, the driver application, command line processor, test suite, lexer, parser, etc. are not included in this document. The full source and documentation for the system is available for the following URL:

`http://www.cs.uvm.edu/~dvanhorn/trace/`

Instructions on building and running the analysis, command line options, and description of

the concrete syntax of λ_{trace} are given in the `README` file. A number of example programs, their inferred types, and the results of applying the various trace effect transformations are included in the `EXAMPLES` file.

4.2 Description of source code

The files included in the appendix are described as follows:

- `traceast.ml`

  This file defines the language of abstract syntax trees for expressions in λ_{trace} as described in Section 2.

- `tracetype.ml`

  This file defines the language of types and trace effects as described in Section 3.

- `traceinfer.ml`

  This files implements the type and trace effect inference algorithm given in Figure 6. The resulting constrained type and effect can either be unified to obtain a type derivable in the weakening system of Section 3, or can be closed according the subtype interpretation of constraints and checked for consistency to obtain a type derivable in the subtyping system of (Skalka, Smith, and Van Horn 2005). Both of these interpretations are given by the following two files.

- `traceweaken.ml`

  This file defines the constraint inference algorithm for the weaken system, unification, and the *MGS* algorithm as described in Section 3.

- `tracesubtype.ml`

  This file define the subtyping interpretation of the flows to relation for obtaining the subtyping inference system described in (Skalka, Smith, and Van Horn 2005).

- `tracetransform.ml`

  This file defines the effect transformation algorithms as described in Section 3.3.2 and (Skalka, Smith, and Van Horn 2005).

The implementation proved to be an indispensable tool for the theoretical development. It helped identify several problems during the development of these algorithms, and was a useful feedback tool while developing the theoretical framework of trace effect analysis. It was used extensively during the preparation of the examples used throughout this document.

5 Conclusion

This thesis has described an automated method for performing trace effect analysis, demonstrated its implementation, and proved that programs analyzed under this method are safe; they obey their temporal specifications for all possible executions. These temporal properties express the well-formedness of program events and are expressive enough to capture many temporal program correctness properties. The analysis is realized in the form of a programming type system and automation is accomplished through polymorphic type and effect inference techniques. All the needed algorithms have been implemented for a prototype functional programming language.

References

Abadi, M. and C. Fournet (2003, feb). Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS'03)*. 13

Alpern, B. and F. B. Schneider (1984). Defining liveness. Technical report, Ithaca, NY, USA. 13

Amtoft, T., F. Nielson, and H. R. Nielson (1999). *Type and Effect Systems*. Imperial College Press.
`http://www2.imm.dtu.dk/~riis/WebDesign/tba.html`. 12, 13

Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics*. Amsterdam: North Holland. 10

Burkart, O., D. Caucal, F. Moller, and B. Steffen (2001). Verification on infinite structures. In S. S. J. Bergstra, A. Pons (Ed.), *Handbook on Process Algebra*. North-Holland.
`http://www.irisa.fr/galion/caucal/HANDBOOK.ps`. 12, 25

Burn, G., C. Hankin, and S. Abramsky (1986). Strictness analysis for higher order functions. *Science of Computer Programming 7*, 249–278. 9

Cousot, P. and R. Cousot (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Los Angeles, California, pp. 238–252. ACM Press, New York, NY.
`http://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml`. 7

Cousot, P. and R. Cousot (1994, May). Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proceedings of the 1994 International Conference on Computer Languages*, Toulouse, France, pp. 95–112. IEEE Computer Society Press, Los Alamitos, California.
`http://www.di.ens.fr/~cousot/COUSOTpapers/ICCL94.shtml`. 9

Damas, L. and R. Milner (1982). Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 207–212. ACM Press. 44

Damas, L. M. M. (1985). Type assignment in programming languages. Technical Report CST-33-85, University of Edinburgh, Department of Computer Science. Ph.D. Thesis. 44

Edjlali, G., A. Acharya, and V. Chaudhary (1998). History-based access control for mobile code. In *ACM Conference on Computer and Communications Security*, pp. 38–48. 13

Eifrig, J., S. Smith, and V. Trifonov (1995). Type inference for recursively constrained types and its application to OOP. Volume 1.
`http://www.elsevier.nl/locate/entcs/volume1.html`. 15, 35

Esparza, J. (1994, April). On the decidability of model checking for several mu-calculi and petri nets. In S. Tison (Ed.), *Proceedings of the 19th International Colloquium on Trees in Algebra and Programming (CAAP '94) Trees in Algebra and Programming*, Lecture Notes in Computer Science, Edinburgh, U.K. Springer-Verlang.
`http://www.lfcs.inf.ed.ac.uk/reports/93/ECS-LFCS-93-274/ECS-LFCS-93-274.ps`. 11

Gentzen, G. (1935). Untersuchungen über das logische schliessen. *Mathmatische Zeitschrift 39*, 176–210,405–431. Translated under the title *Investigations into Logical Deductions* in (Szabo 1969). 25

Hankin, C. (2004). *An Introduction to Lambda Calculi for Computer Scientists*, Volume 2. King's College London.
`http://www.dcs.kcl.ac.uk/kcl-publications/comp/vol2.html`. 10

Hindley, J. R. (1969). The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 29–60. 15

Holzmann, G. J. (2003). Trends in software verification. In K. Araki, S. Gnesi, and D. Mandrioli (Eds.), *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, Lecture Notes in Computer Science, pp. 40–50. Springer-Verlang.
`http://spinroot.com/gerard/pdf/fme03.pdf`. 6

Holzmann, G. J. and M. H. Smith (2001). Software model checking: extracting verification models from source code. *Software Testing, Verification & Reliability 11*(2), 65–79.
`http://spinroot.com/gerard/pdf/fortepstv99.pdf`. 6

Jones, N. D. and F. Nielson (1995). Abstract interpretation: a semantics-based tool for program analysis. pp. 527–636. 7

Kozen, D. (1983, December). Results on the propositional mu-calculus. *Theoretical Computer Science 27*, 333–354. 11

Kupferman, O. and M. Y. Vardi (2001). Model checking of safety properties. *Formal Methods in Systems Design 19*(3), 291–314. 5, 6, 12

Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering 3*(2), 125–143. 13

Lee, O. and K. Yi (1998). Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions of Programming Languages and Systems (TOPLAS) 20*(4), 707–723.
`http://ropas.snu.ac.kr/lib/dock/LeYi1998.pdf`. 44

Leroy, X. (2004, July). The Objective Caml system release 3.08: Documentation and user's manual.
`http://caml.inria.fr/pub/docs/manual-ocaml/`. 72

Melton, A., D. A. Schmidt, and G. E. Strecker (1986). Galois connections and computer science applications. In *Proceedings of a tutorial and workshop on Category theory and computer programming*, New York, NY, USA, pp. 299–312. Springer-Verlag New York, Inc. 8

Milner, R. (1978, August). A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 348–375. 15

Milner, R., M. Tofte, R. Harper, and D. MacQueen (1997). *The Definition of Standard ML (Revised)*. MIT-Press. 5, 16

Naraschewski, W. and T. Nipkow (1998). Type inference verified: Algorithm W in Isabelle/HOL. In E. Giménez and C. Paulin-Mohring (Eds.), *Types for Proofs and Programs: Intl. Workshop TYPES '96*, Volume 1512, pp. 317–332. 44

Nielson, F. and H. R. Nielson (1999). Type and effect systems. In *E. R. Olderog and B. Steffen, editors, Correct System Design*, Volume 1710 of *Lecture Notes in Computer Science*, pp. 114–136. Springer Verlang.
`http://www.cs.ucla.edu/~palsberg/tba/papers/nielson-nielson-csd99.pdf`. See Jens Palsberg's bibliography on "Type-Based Analysis and Applications" for this and related papers,
`http://www.cs.ucla.edu/~palsberg/tba/`. 12

Nielson, F., H. R. Nielson, and C. Hankin (1999). *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.
`http://www2.imm.dtu.dk/~riis/PPA/ppa.html`. 7

Paterson, M. S. and M. N. Wegman (1976). Linear unification. In *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, New York, NY, USA, pp. 181–186. ACM Press. 15

Pfenning, F. (2002). Linear logic.
`http://www-2.cs.cmu.edu/~fp/courses/linear/handouts/linear.pdf`. 15

Pierce, B. C. (2002). *Types and Programming Languages*. The MIT Press. 38, 53

President's Information Technology Advisory Committee (1999). Report to the President: Information Technology Research: Investing in Our Future.
`http://www.itrd.gov/pitac/report/pitac_report.pdf`. 4

Rémy, D. (1992). Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, New-York, pp. 66–75. ACM press.
`ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/lfp92.ps.gz`. 67

Robinson, J. A. (1971). Computational logic: The unification computation. *Machine Intelligence*, 63–72. 15

Schmidt, D. A. (1998a). Data flow analysis is model checking of abstract interpretations. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, pp. 38–48. ACM Press.
`http://www.cis.ksu.edu/~schmidt/papers/dfa.ps.gz`. 9

Schmidt, D. A. (1998b). Trace-Based Abstract Interpretation of Operational Semantics. *Lisp and Symbolic Computation 10*(3), 237–271.
`http://www.cis.ksu.edu/~schmidt/papers/aiosh.ps.gz`. 9

Schmidt, D. A. and B. Steffen (1998). Program analysis as model checking of abstract interpretations. In *SAS '98: Proceedings of the 5th International Symposium on Static Analysis*, Volume 1503 of *Lecture Notes in Computer Science*, London, UK, pp. 351–380. Springer-Verlag.
`http://www.cis.ksu.edu/~schmidt/papers/paperneu9.ps.gz`. 8

Schneider, F. B. (2000). Enforceable security policies. *Information and System Security 3*(1), 30–50. 1

Shivers, O. G. (1991). *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda*. Ph. D. thesis, Pittsburgh, PA, USA.
`http://www.cc.gatech.edu/~shivers/papers/cmu/diss.ps`. 9

Skalka, C. and S. Smith (2004a, November). History effects and verification. In *Asian Programming Languages Symposium*.
`http://www.cs.uvm.edu/~skalka/skalka-pubs/skalka-smith-aplas04.pdf`. 1, 11, 12, 15, 16, 21, 29, 36, 48, 64

Skalka, C. and S. Smith (2004b). History types and verification. Extended manuscript,
`http://www.cs.uvm.edu/~skalka/skalka-smith-tr04.ps`. 34

Skalka, C., S. Smith, and D. Van Horn (2005, January). A Type and Effect System for Flexible Abstract Interpretation of Java (Extended Abstract). In *Proceedings of the ACM Workshop on Abstract Interpretation of Object Oriented Languages*, Electronic Notes in Theoretical Computer Science.
`http://www.cs.uvm.edu/~skalka/skalka-pubs/skalka-smith-vanhorn-aiool05.pdf`. 17, 72, 73, 74

Smith, S. (2002). *Programming Languages*.
`http://www.cs.jhu.edu/~scott/plbook/`. 72

Szabo, M. E. (Ed.) (1969). *The Collected Papers of Gerhard Gentzen*. Amsterdam: North-Holland Publishing Co. 77

United States Computer Emergency Readiness Team (2005). US-Cert Vulnerability Notes Database.
`http://www.kb.cert.org/vuls/`. 5

Wallach, D. S. and E. Felten (1998, May). Understanding Java stack inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*. 13, 72

Wells, J. B. (1999). Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic 98*(1–3), 111–156.
`http://www.macs.hw.ac.uk/~jbw/papers/f-undecidable-APAL.ps.gz`. 16

Wright, A. K. and M. Felleisen (1994). A syntactic approach to type soundness. *Information and Computation 115*(1), 38–94. 5

A Source Code

A.1 traceast.ml

```
type expr =
  | Seq of expr * expr
  | App of expr * expr
  | Event of label * expr
  | Check of label * expr
  | If of expr * expr * expr
  | Let of evar * expr * expr
  | Var of evar
  | Fix of evar * evar * expr
  | Const of label
  | Bool of bool
  | Unit
  | Not
  | Or
  | And
  | Try of expr * (label * expr) list
  | Throw
 and label = string
 and evar = string
```

A.2 tracetype.ml

```
open Traceast;;

(** {i Singletons.} *)
type singleton =
    SVar of label
  | SConst of label

(** {i Trace Effects.} *)
type heffect =
    HEv of label * singleton
  | Choice of heffect * heffect
  | HVar of label
  | Mu of label * heffect
  | Sequence of heffect * heffect
  | Throw
  | Catch of heffect
  | Epsilon

(**
   {i Types.}

   \[\[[SingletonSet(s)]\]\] = \{s\}, whereas \[\[[Singleton(s)]\]\] = s.
*)
type htype =
```

```
    TVar of tvar
  | SingletonSet of singleton
  | TBool
  | TUnit
  | Arrow of htype * heffect * htype
  | Singleton of singleton
  | HEffect of heffect
and tvar = string                 (** {i Type variables.} *)

(** {i Type solutions.} *)
type solution = (htype * htype) list

module Cset =
  Set.Make(struct type t = (htype * htype) let compare = compare end)

(* Should change comparison to enforce that a Vset is comprised only of
   values.
*)
module Vset =
  Set.Make(struct type t = htype let compare = compare end)

module Hset =
  Set.Make(struct type t = heffect let compare = compare end)

module Rset =  (* Rset for "recursors" set *)
  Set.Make(struct type t = (heffect * label) let compare = compare end)

include Cset

(** {i Constraint Sets.} *)
type constraint_set = t

(**
   {i Constrained Type Schemes.}

   [(beta,tau,c,hc)] binds all sorts of type variables in beta in type
   tau, constraint set C, and effect constraint set HC.  All types in list
   beta must be of the form:
   - [TVar t]
   - [HEffect (HVar h)]
   - [Singleton (SVar s)]
*)
type constrained_type_scheme =
    (htype list * htype * constraint_set * constraint_set)

(** {i Type Environments.}  Given in Figure 3. *)
type type_env = (evar * constrained_type_scheme) list

let empty_type_env : type_env = []
```

A.3 traceinfer.ml

```
open Traceast;;
open Tracetype;;

exception SubstError of htype * htype * htype

let rec subst_heffect heff heff' h =
  let HVar hv = h in
  match heff with
    HVar h' when hv=h' -> heff'
  | Choice(h1,h2) ->
      Choice((subst_heffect h1 heff' h), (subst_heffect h2 heff' h))
  | Sequence(h1,h2) ->
      Sequence((subst_heffect h1 heff' h), (subst_heffect h2 heff' h))
  | Mu(h',h1) ->
      Mu((if hv=h' then let HVar hv' = heff' in hv' else h'),
 (subst_heffect h1 heff' h))
  | _ -> heff


(**
   {i Type substitution.}

   Substitutes type tau' for variable beta in type tau.  The type of tau'
   must respect the sort of variable beta.

   \[\[[subst_htype tau tau' beta]\]\] = tau\[tau'/beta\]

   Beta is a variable of any sort.  It can be of the following form:
   - [TVar t]
   - [HEffect (HVar h)]
   - [Singleton (SVar s)]

   When beta is of the form [TVar t], tau' can be any htype which is {b
   not} a [Singleton] or [HEffect].

   When beta is of the form [HEffect (HVar h)], tau' must of the form
   [HEffect h].

   When beta is of the form [Singleton (SVar s)], tau' must be of the form
   [Singleton s].

*)
let rec subst_htype =
  let rec subst_singleton heff s' s =
    match heff with
    | HEv(l,sing) when sing = s -> HEv(l,s')
    | Choice(h1,h2) ->
Choice((subst_singleton h1 s' s), (subst_singleton h2 s' s))
```

```
    | Sequence(h1,h2) ->
Sequence((subst_singleton h1 s' s), (subst_singleton h2 s' s))
    | Mu(h,h1) ->
Mu(h,(subst_singleton h1 s' s))
    | _ -> heff
  in
  fun tau tau' b ->
    match (tau',b) with
    | Singleton _, TVar t -> raise (SubstError(tau,tau',b))
    | HEffect _, TVar t -> raise (SubstError(tau,tau',b))
    | tau', TVar t ->
(match tau with
| TVar t' when t=t' -> tau'
| Arrow(t1,h,t2) ->
    Arrow((subst_htype t1 tau' b),
  h,
  (subst_htype t2 tau' b))
| _ -> tau)

    | HEffect heff, HEffect (HVar hv) ->
(match tau with
| Arrow(t1,h,t2) ->
    Arrow((subst_htype t1 tau' b),
  (subst_heffect h heff (HVar hv)),
  (subst_htype t2 tau' b))
| HEffect h -> HEffect (subst_heffect h heff (HVar hv))
| _ -> tau)
    | Singleton sing, Singleton (SVar s) ->
(match tau with
| SingletonSet (SVar s') when s=s' -> SingletonSet sing
| Singleton (SVar s') when s=s' -> Singleton sing
| Arrow(t1,h,t2) ->
    Arrow((subst_htype t1 tau' b),
  (subst_singleton h sing (SVar s)),
  (subst_htype t2 tau' b))
| HEffect h ->
    HEffect (subst_singleton h sing (SVar s))
| _ -> tau)
    | _ -> raise (SubstError(tau,tau',b))

(**
   {i Set mapping.}

   [set_map f s] returns \{f(x) | x in S\}.
*)
let set_map f s = fold (fun x s -> (add (f x) s)) s empty

(**
   {i Set choice.}
```

```
   [set_choose s] returns (x,S\\{x\}) for some x in S.
*)
let set_choose s = let e = choose s in (e, remove e s)

(**
   {i Constraint Set substitution.}

   Substitutes type tau for type variable t in constraint set c.  This
   function extends type substitution to constraint sets in the obvious
   manner, ie. the function returns the constraint set obtained by
   substituting tau for t in each type in C.

   \[\[[subst_cset c tau t]\]\] = C\[tau/t\]
*)
let subst_cset (c : constraint_set) tau t =
  set_map (fun(t1,t2) -> ((subst_htype t1 tau t), (subst_htype t2 tau t))) c

let compose_map s1 s2 = s1@s2

let apply_map soln tau =
  List.fold_right (fun (tau',t) tau -> subst_htype tau tau' t) soln tau

let apply_map_cset soln (c : constraint_set) =
  set_map (fun (t1,t2) -> ((apply_map soln t1), (apply_map soln t2))) c

(**
   {i Free variables}

   [fv_htype tau] returns fv(tau).

   [fv_cset c] extends fv to constraint sets.
*)
let rec fv_htype t =
   match t with
     TVar _ as t -> Vset.singleton t
   | SingletonSet(SVar s) -> Vset.singleton (Singleton (SVar s))
   | Arrow(t1,h,t2) ->
       Vset.union
 (Vset.union
    (fv_htype t1)
    (fv_htype (HEffect h)))
 (fv_htype t2)
   | Singleton(SVar _) as t -> Vset.singleton t
   | HEffect(HVar _) as t -> Vset.singleton t
   | HEffect(HEv (_,(SVar s))) -> Vset.singleton (Singleton (SVar s))
   | HEffect(Sequence(h1,h2)) ->
       Vset.union
 (fv_htype (HEffect h1))
 (fv_htype (HEffect h2))
   | HEffect(Choice(h1,h2)) ->
```

```
       Vset.union
 (fv_htype (HEffect h1))
 (fv_htype (HEffect h2))
   | HEffect(Mu(h,h1)) ->
       Vset.filter
 ((=) (HEffect(HVar h)))
 (fv_htype (HEffect h1))
   | _ -> Vset.empty

let fv_cset (c : constraint_set) =
  fold
    (fun (t1,t2) fv ->
      (Vset.union fv (Vset.union (fv_htype t1) (fv_htype t2))))
    c
    Vset.empty

let fv_type_env (gamma : type_env) =
  List.fold_right
    (fun (x,(b,t,c,hc)) fv ->
      Vset.union
(Vset.diff
   (Vset.union
      (Vset.union (fv_htype t) (fv_cset c))
      (fv_cset hc))
   Vset.empty)
 (*  (Vset.fold Vset.add b Vset.empty)) *)
fv)
    gamma
    Vset.empty

(**
   {i Bounds.}

   [bounds h c] constructs a history effect H | ... | H from all
   constraints of the form "H flows into h" in C.
*)
let bounds b (c : constraint_set) =
  match b with
    HEffect(HVar _) ->
      let x =
fold
  (fun flowsto choices ->
    match flowsto with
    | (((HEffect he) as h), b') when b=b' ->
(match choices with
| None -> Some he
| Some choices -> Some(Choice(he,choices)))
    | _ -> choices)
  c
  None
```

```
      in
      (match x with None -> Epsilon | Some c -> c)

(**
   bounds_c(ev_x(a),C) =  ev_x(c_1) | ... |  ev_x(c_n) for all c <: a in C.
*)
let bounds_c ev (c : constraint_set) =
  match ev with
    HEv(l,(SVar a)) ->
      let x =
fold
  (fun flowsto choices ->
    match flowsto with
    | ((Singleton ((SConst _) as c)), (Singleton (SVar a'))) when a=a' ->
(match choices with
| None -> Some (HEv(l,c))
| Some choices -> Some(Choice((HEv(l,c)),choices)))
    | _ -> choices)
  c
  None
      in
      (match x with None -> Epsilon | Some c -> c)


(*
let rec mgs_heffect (c : constraint_set) : solution =
  let c' = filter
      (function
| (HEffect _), (HEffect(HVar _)) -> true
| _ -> false)
      c in
  if is_empty c' then []
  else
    let (_,h) = choose c' in
    let (HEffect (HVar hv)) = h in

    compose_map
      (mgs_heffect
 (apply_map_cset p
    (diff
       c
       (filter (fun (he,h') -> h'=h) c'))))
      p
*)

let fresh_hvar =
  let i = ref 0 in
  fun () ->
    i := !i+1;
    HVar ("h_" ^ (string_of_int !i) ^ "")
```

```
let rec mgs_heffect (c : constraint_set) : solution =
  let c' = filter
      (function
| (HEffect _), (HEffect(HVar _)) -> true
| _ -> false)
      c in
  if is_empty c' then []
  else
    let (_,h) = choose c' in
    let (HEffect (HVar hv)) = h in
    (* This version was found to return non-MG substitutions. 2/28/05.
    let p = [(HEffect(Mu(hv,(bounds h c)))),h] in *)
    let p = [(HEffect(Choice((Mu(hv,(bounds h c))), fresh_hvar()))),h] in
    compose_map
      (mgs_heffect
 (apply_map_cset p
    (diff
       c
       (filter (fun (he,h') -> h'=h) c'))))
      p


let is_variable = function
  | TVar _ -> true
  | Singleton(SVar _) -> true
  | HEffect(HVar _) -> true
  | _ -> false

exception RecursiveConstraint

let occurs t tau = (Vset.mem t (fv_htype tau))

let rec unify (c : constraint_set) : solution =
  if is_empty c then []
  else let (x, c') = set_choose c in
  match x with
  | tau,tau' when tau=tau' -> unify c'
  | t,tau when is_variable t ->
      if occurs t tau then raise RecursiveConstraint else
      compose_map (unify (apply_map_cset [tau,t] c')) [tau,t]
  | tau,t when is_variable t ->
      if occurs t tau then raise RecursiveConstraint else
      compose_map (unify (apply_map_cset [tau,t] c')) [tau,t]
  | SingletonSet s1, SingletonSet s2 ->
      unify (add (Singleton s1, Singleton s2) c')
  | Arrow(t1,h1,t2), Arrow(t1',h2,t2') ->
      unify (add (HEffect h1, HEffect h2) (add (t1,t1') (add (t2',t2) c')))

let mgs (c,hc) =
```

```
  let p = unify c in
  compose_map
    (mgs_heffect (apply_map_cset p hc))
    p

let fresh_tvar =
  let i = ref 0 in
  fun () ->
    i := !i+1;
    TVar ("t_" ^ (string_of_int !i) ^ "")

let fresh_svar =
  let i = ref 0 in
  fun () ->
    i := !i+1;
    SVar ("'a_" ^ (string_of_int !i) ^ "")

let rename (b,t,c,hc) =
  List.fold_right
    (fun x k ->
      let (t,c,hc) = k in
      let t' =
(match x with
| TVar _ -> fresh_tvar()
| HEffect (HVar _) -> HEffect(fresh_hvar())
| Singleton(SVar _) -> Singleton(fresh_svar()))
      in
      ((subst_htype t t' x), (subst_cset c t' x), (subst_cset hc t' x)))
    b
    (t,c,hc)

(**
   {i Type Inference for Weaken.}
*)
let rec infer (g,e) : (heffect * htype * constraint_set * constraint_set) =
  match e with
  | Var x ->
      let (t,c,hc) = rename(List.assoc x g) in
      Epsilon,t,c,hc

  | Event(l,e) ->
      let (h,t,c,hc) = infer (g,e) in
      let a = fresh_svar() in
      (Sequence(h,HEv(l,a))), TUnit, (add (t,(SingletonSet a)) c), hc

  | Check(l,e) ->
      let (h,t,c,hc) = infer (g,e) in
      let a = fresh_svar() in
      (Sequence(h,HEv("\\phi_{"^l^"}",a))), TUnit,
      (add (t,(SingletonSet a)) c), hc
```

```
  | If(e1,e2,e3) ->
      let (h1,t1,c1,hc1) = infer (g,e1) in
      let (h2,t2,c2,hc2) = infer (g,e2) in
      let (h3,t3,c3,hc3) = infer (g,e3) in
      let t = fresh_tvar() in
      (Sequence(h1,(Choice(h2,h3)))), t,
      List.fold_right add
[(t1,TBool); (t2,t); (t3,t)]
(union (union c1 c2) c3),
      (union (union hc1 hc2) hc3)

  | App(e1,e2) ->
      let (h1,t1,c1,hc1) = infer (g,e1) in
      let (h2,t2,c2,hc2) = infer (g,e2) in
      let t = fresh_tvar() in
      let h = fresh_hvar() in
      (Sequence(h1,(Sequence(h2,h)))), t,
      (add (t1,(Arrow(t2,h,t)))
 (add ((Arrow(t2,h,t)),t1)
    (union c1 c2))),
      (union hc1 hc2)

  | Fix(z,x,e) ->
      let t = fresh_tvar() in
      let t'= fresh_tvar() in
      let h = fresh_hvar() in
      let g = (x,([],t,empty,empty))::(z, ([],Arrow(t,h,t'),empty,empty))::g in
      let (heff,tau,c,hc) = infer (g,e) in
      Epsilon, (Arrow(t,h,tau)),
      (add (tau,t') c),
      (add ((HEffect heff),(HEffect h)) hc)

  | Let(x,v,e) ->
      let (Epsilon,t',c',hc') = infer (g,v) in
      let beta = Vset.elements
  (Vset.diff
     (Vset.union
(Vset.union (fv_htype t') (fv_cset c'))
(fv_cset hc'))
     (fv_type_env g)) in
      let g = (x,(beta,t',c',hc'))::g in
      let (h,t,c,hc) = infer (g,e) in
      h, t, (union c' c), (union hc' hc)

  | Seq(e1,e2) ->
      let (h1,t1,c1,hc1) = infer (g,e1) in
      let (h2,t2,c2,hc2) = infer (g,e2) in
      Sequence(h1,h2), t2, (union c1 c2), (union hc1 hc2)
```

```
  | Const c -> Epsilon, (SingletonSet (SConst c)), empty, empty
  | Bool _ -> Epsilon, TBool, empty, empty
  | Unit   -> Epsilon, TUnit, empty, empty
  | And -> Epsilon, Arrow(TBool, Epsilon, Arrow(TBool, Epsilon, TBool)),
      empty, empty
  | Or  -> Epsilon, Arrow(TBool, Epsilon, Arrow(TBool, Epsilon, TBool)),
      empty, empty
  | Not -> Epsilon, Arrow(TBool, Epsilon, TBool), empty, empty

(*
   {i Type Inference for Subsumption.}
*)
let rec infer_sub (g,e) : (heffect * htype * constraint_set) =
  match e with
  | Var x ->
      let (t,c,_) = rename(List.assoc x g) in
      Epsilon,t,c

  | Event(l,e) ->
      let (h,t,c) = infer_sub (g,e) in
      let a = fresh_svar() in
      (Sequence(h,HEv(l,a))), TUnit, (add (t,(SingletonSet a)) c)

  | Check(l,e) ->
      let (h,t,c) = infer_sub (g,e) in
      let a = fresh_svar() in
      (Sequence(h,HEv("\\phi_{"^l^"}",a))), TUnit,
      (add (t,(SingletonSet a)) c)

  | If(e1,e2,e3) ->
      let (h1,t1,c1) = infer_sub (g,e1) in
      let (h2,t2,c2) = infer_sub (g,e2) in
      let (h3,t3,c3) = infer_sub (g,e3) in
      let t = fresh_tvar() in
      (Sequence(h1,(Choice(h2,h3)))), t,
      List.fold_right add
[(t1,TBool); (t2,t); (t3,t)]
(union (union c1 c2) c3)

  | App(e1,e2) ->
      let (h1,t1,c1) = infer_sub (g,e1) in
      let (h2,t2,c2) = infer_sub (g,e2) in
      let t = fresh_tvar() in
      let h = fresh_hvar() in
      (Sequence(h1,(Sequence(h2,h)))), t,
      (add (t1,(Arrow(t2,h,t)))
 (union c1 c2))

  | Fix(z,x,e) ->
      let t = fresh_tvar() in
```

```
      let t'= fresh_tvar() in
      let h = fresh_hvar() in
      let g = (x,([],t,empty,empty))::(z, ([],Arrow(t,h,t'),empty,empty))::g in
      let (heff,tau,c) = infer_sub (g,e) in
      Epsilon, (Arrow(t,h,tau)),
      (add (tau,t')
 (add ((HEffect heff),(HEffect h)) c))

  | Let(x,v,e) ->
      let (Epsilon,t',c') = infer_sub (g,v) in
      let beta = Vset.elements
  (Vset.diff
     (Vset.union (fv_htype t') (fv_cset c'))
     (fv_type_env g)) in
      let g = (x,(beta,t',c',empty))::g in
      let (h,t,c) = infer_sub (g,e) in
      h, t, (union c' c)

  | Seq(e1,e2) ->
      let (h1,t1,c1) = infer_sub (g,e1) in
      let (h2,t2,c2) = infer_sub (g,e2) in
      Sequence(h1,h2), t2, (union c1 c2)

  | Const c -> Epsilon, (SingletonSet (SConst c)), empty
  | Bool _ -> Epsilon, TBool, empty
  | Unit   -> Epsilon, TUnit, empty
  | And -> Epsilon, Arrow(TBool, Epsilon, Arrow(TBool, Epsilon, TBool)), empty
  | Or  -> Epsilon, Arrow(TBool, Epsilon, Arrow(TBool, Epsilon, TBool)), empty
  | Not -> Epsilon, Arrow(TBool, Epsilon, TBool), empty


let rec find_fix equal f =
  let rec recur x =
    let y = (f x) in
    if equal y x then x
    else recur y
  in
  recur

let close_step c =
  fold
    (fun (t1,t2) set ->
      union
(set_map
   (function (t2,t3) -> (t1,t3))
   (filter (function t2',t3 -> t2=t2') c))
(match (t1,t2) with
| (Arrow(t1,h,t2), Arrow(t1',h',t2')) ->
    (add (t1',t1)
       (add (t2,t2')
```

```
  (add ((HEffect h), (HEffect h')) set)))
| (SingletonSet t1), (SingletonSet t2) ->
    (add ((Singleton t1),(Singleton t2)) set)
| _ -> set))
    c
    c

(* At each iteration, map the closure rules across the set of contraints.
   Iterate until a fixed point is reached.
*)
let close = find_fix equal close_step

let rec hextract (h,hs,c) =
  match h with
  | Epsilon -> Epsilon
  | HEv(l,a) -> bounds_c h c
  | HVar _ when Vset.mem (HEffect h) hs -> h
  | HVar hv -> Mu(hv, (hextract((bounds (HEffect h) c), (Vset.add (HEffect h) hs), c)))
  | Sequence(h1,h2) -> Sequence(hextract(h1,hs,c),hextract(h2,hs,c))
  | Choice(h1,h2)   ->   Choice(hextract(h1,hs,c),hextract(h2,hs,c))

let rec consistent_p c =
  for_all
    (function
      | tau,tau' when tau=tau' -> true
      | SingletonSet _,SingletonSet _ -> true
      | HEffect _,HEffect _ -> true
      | ((TVar _,tau) | (tau, TVar _)) ->
  (match tau with
  | Singleton _ -> false
  | HEffect _ -> false
  | _ -> true)
      | (((Singleton (SVar _)),(Singleton _)) | ((Singleton _), (Singleton (SVar _)))) -> true
      | Arrow(_,_,_), Arrow(_,_,_) -> true
      | _ -> false)
    c
```

A.4 tracetransform.ml

```
open Tracetype;;

(**
   {i Stackify transformation.}
*)
let rec stackify = function
  | Epsilon -> Epsilon
  | Sequence(Epsilon,h) -> stackify h
  | Sequence(HEv(l,c), h) -> Sequence((HEv(l,c)), stackify h)
```

```
  | Sequence(HVar hv, h) -> Choice((HVar hv), stackify h)
  | Sequence(Mu(hv,h1),h2) -> Choice((Mu(hv, stackify h1)), stackify h2)
  | Sequence(Choice(h1,h2),h) -> Choice(stackify (Sequence (h1,h)),
stackify (Sequence (h2,h)))
  | Sequence(Sequence(h1,h2),h3) -> stackify(Sequence(h1,(Sequence(h2,h3))))
  | h -> stackify (Sequence(h,Epsilon))

let rec simplify_heffect = function
  | (Sequence(Epsilon,h)
  |  Sequence(h,Epsilon)) ->
      simplify_heffect h
  | Sequence(h,h') ->
      Sequence ((simplify_heffect h),(simplify_heffect h'))
  | Choice(h,h') ->
      let h = simplify_heffect h  in
      let h'= simplify_heffect h' in
      if h=h' then h
      else Choice(h,h')
  | Mu(l,h) when not (Vset.mem (HEffect (HVar l)) (fv_htype (HEffect h))) ->
      simplify_heffect h
  | Mu(l,h) -> Mu(l,(simplify_heffect h))
  | h -> h

let rec simplify_htype = function
  | Arrow(t1,h,t2) ->
      Arrow((simplify_htype t1), (simplify_heffect h), (simplify_htype t2))
  | HEffect(h) -> HEffect(simplify_heffect h)
  | t -> t

let simplify = find_fix (=) simplify_htype

(**
   {i Set choice. (specialized for Hsets)}

   [hset_choose s] returns (x,S\\{x\}) for some x in S.
*)
let rec hset_choose s = let e = Hset.choose s in (e, Hset.remove e s)

let rec join hset =
  let (h, s) = hset_choose hset in
  if Hset.is_empty s then h
  else Choice(h, (join s))

(**
   {i Cartesian sequencing construct}

   For sequencing pairs of history effect sets.

   [seq s1 s2] returns \{h1; h2 | h1 in S1, h2 in S2\}.
*)
```

```
let seq s1 s2 =
  let f h s = Hset.fold
      (fun h' s' -> Hset.add (Sequence (h,h')) s') s Hset.empty
  in
  Hset.fold
    (fun h s -> Hset.union (f h s2) s) s1 Hset.empty


(**
   {i Cartesian sequencing construct}

   For sequencing a history effect set and a recursors set.

   [rseq s r] returns \{(h1; h2, h) | h1 in S, h2,h in R\}.
*)
let rseq s r =
  let f h r = Rset.fold
      (fun (h',hv) r -> Rset.add (Sequence (h,h'), hv) r) r Rset.empty
  in
  Hset.fold
    (fun h r' -> Rset.union (f h r) r') s Rset.empty


(**
    Compute the set of variables in a history effect.
*)
let rec hvs = function
  | Epsilon  -> Hset.empty
  | HEv(_,_) -> Hset.empty
  | Throw    -> Hset.empty
  | HVar _ as h -> Hset.singleton h
  | Choice(h1,h2)   -> Hset.union (hvs h1) (hvs h2)
  | Sequence(h1,h2) -> Hset.union (hvs h1) (hvs h2)
  | Mu(_,h) -> hvs h
  | Catch h -> hvs h

let rset_map f s = Rset.fold (fun x s -> (Rset.add (f x) s)) s Rset.empty
let hset_map f s = Hset.fold (fun x s -> (Hset.add (f x) s)) s Hset.empty

(**
   {i Exnixation transformation.}

   Given in email by Chris, 11/15/04.
*)

(* heffect -> (Hset s, Hset t, Rset r) *)

let rec exnize = function
    Epsilon -> (Hset.singleton Epsilon, Hset.empty, Rset.empty)
  | HEv(_,_) as ev -> (Hset.singleton ev, Hset.empty, Rset.empty)
```

```
  | Throw -> (Hset.empty, Hset.singleton Throw, Rset.empty)
  | HVar hv as h -> (Hset.singleton h, Hset.empty,
     Rset.singleton (Epsilon, hv))
  | Choice(h1,h2) ->
      let s1,t1,r1 = exnize h1 in
      let s2,t2,r2 = exnize h2 in
      (Hset.union s1 s2, Hset.union t1 t2, Rset.union r1 r2)

  | Sequence(h1,h2) ->
      let s1,t1,r1 = exnize h1 in
      let s2,t2,r2 = exnize h2 in
      (seq s1 s2, (Hset.union t1 (seq s1 t2)), Rset.union r1 (rseq s1 r2))

  | Catch h ->
      let s,t,r = exnize h in
      (Hset.union s t, Hset.empty, r)

  | Mu(hv,h) ->
      let s,t,r = exnize h in
      if Hset.is_empty s then
let r' = Rset.filter
    (fun (h,_) -> (not (Hset.exists ((=)(HVar hv)) (hvs h)))) r in
let rh = Rset.filter (fun (h,hv') -> hv' = hv) r' in
let sh = Rset.fold
    (fun (h,hv) s -> (Hset.add (Sequence (h,(HVar hv))) s))
    rh Hset.empty
in
let t''= hset_map (fun h -> Mu(hv, join (Hset.add h sh))) t in
(Hset.empty, t'', Rset.diff r' rh)
      else
let hs = Mu(hv, join s) in
let r' = rset_map
    (fun (h, hv') -> ((subst_heffect h hs (HVar hv)), hv')) r
in
let rh = Rset.filter (fun (h,hv') -> hv' = hv) r' in
let sh = Rset.fold
    (fun (h,hv) s -> (Hset.add (Sequence (h,(HVar hv))) s))
    rh Hset.empty
in
let t' = hset_map
    (fun h ->
      Mu(hv, join (Hset.add (subst_heffect h hs (HVar hv)) sh)))
    t
in
(Hset.singleton hs, t', Rset.diff r' rh)

exception Recursors_exn

let exnize_top h =
  let s,t,r = exnize h in
```

```
if Rset.is_empty r then
  join (Hset.union s t)
else
  raise Recursors_exn
```