Types for Programming Language-Based Security

by

Christian Skalka

A thesis submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

June, 2002

Abstract

Programming language-based security provides applications programmers with greater control and flexibility for specifying and enforcing security policies. As computing environments distribute and diversify, and demands for program mobility increase, this power allows programmers to adapt software to developing needs, while still protecting resources. This thesis advocates the use of static type disciplines for expressing and enforcing programming language-based security. We develop type systems for two popular security models: the *Access Control* model with *Stack Inspection*, and the *Object Confinement* model. Type safety proofs demonstrate that these type systems reliably enforce security statically, and imply that certain run-time optimizations may be effected for programs. The declarative nature of our type terms also provide programmers with useful and understandable descriptions of security properties. To formally develop these type systems, a transformational approach is used, where source languages are embedded in a target language, containing sets of atomic elements and associated operations, which is pre-equipped with a sound type system. This target language and type system is developed using the type constraint framework HM(X). The transformational approach and HM(X) both allow the re-use of existing theory and implementations, easing proof effort, inspiring design, and providing greater confidence in the correctness of results.

Advisor: Scott Smith
Readers: Scott Smith, Gregory Hager

Acknowledgements

Thanks to Scott for his patience, guidance, and insight. Thanks to François for sharing his technical virtuosity. Thanks to Susan for seeing it through.

Contents

List of Figures

Introduction

This thesis focuses on static, type-based approaches to programming language-based security. It is an argument for both programming language-based security, and for the use of types as static disciplines to enforce security properties in programming languages (PLs). The argument is made by analysis of two distinct security models—the *access control* and *object confinement* models. We develop formal languages that incorporate these models, reflecting features of real implementations, and then develop type systems for the languages that *statically* enforce security; in particular, soundness results for the type systems imply that run-time checks can safely be removed. In addition to improving efficiency and reliability of implementations, we show that type systems also benefit PL-based security systems by serving as readable descriptions of security policies, making these systems easier to use and understand.

An orthogonal aspect of this thesis is the *methodology* used to develop our type systems. The method allows application of existing results and implementations to novel languages and type systems, a *re-use* that saves significant proof effort. Perhaps even more significantly, the ability to study novel type systems by reflecting on other well-studied and developed systems provides significant insight into the "best" design of the former. We will use a *transformational* approach to type system development, characterized by the semantics-preserving translation of a novel source language into a known target language that comes pre-equipped with a sound type system. This method has been exploited before in other contexts, e.g. to develop a type system for record concatenation [33], and to provide a static analysis of information flow [27]. Here we will use it in the development of type systems for access control and object confinement.

To develop the target language of transformations, we will use the HM(X) framework [23]. HM(X) is intended for a modular definition of languages and type systems. It is pre-equipped with a core, stateful functional calculus and constraint-based type system, and can be instantiated with primitive constants along with a specialized type language and interpretation to easily develop specialized languages. Significantly, any instantiation of the framework automatically obtains both type safety and inference for the language core, by adherence to some basic conditions.

Overall, this thesis provides relevant foundational results for the development of static analyses of programming languages with security features, developed in a manner that conserves proof effort and takes advantage of previous work. We introduce each of the principal threads—PL-based security, type systems and methodology— individually and in more depth below, and cite related work in each case.

Programming Language-Based Security

The term "PL-based security" refers to the incorporation of mechanisms into programming languages, either as primitives or as add-on libraries, allowing direct programmer access to security features of the language implementation. While these mechanisms exist for a variety of purposes, the most prevalent, and the one we'll be concerned with, is to enforce safety of non-local, potentially hostile code in a local, trusted execution environment.

PL-based security is distinct from system-level security mechanisms, e.g. SSL, the details of which are designed to be almost completely transparent at the applications level, where access to these mechanisms is at most a pre-defined API. PL-based security, on the other hand, gives the applications programmer extensive control over security features, to such a degree that sophisticated *policies* may be defined and utilized. As argued in [14], PL-based security is useful because it provides more powerful security abstractions to programmers, leading to more robust code. This argument is gaining credence over time, as the internet and mobile computing devices contribute to the popularity of mobile code. In particular, as the demand for mobile applications increases, so does the need for application developers to directly address the issues that mobility raises—especially issues of trust.

The specific security issue this thesis is concerned with is characterized as the control and protection of *resources*. A requirement of mobile code paradigms is that local systems must provide computational resources such as file systems, memory, clock cycles, etc., for non-local program to consume during execution. Security issue arises because non-local code cannot always be trusted to safely consume these resources. Resources may be abused to a systematically crippling degree both unintentionally, by buggy or poorly designed code, and intentionally, by malicious code. Thus, the PL-based security we are concerned with is intended to protect resources of the local system against abuse by non-local code. In particular, we will be concerned with the two most popular models for

PL-based resource protection, the *access control* and *object confinement* security models.

In the access control model, access to resources is mediated by an access control list (ACL), which associates *principals* with sets of resource *privileges*. Depending on the system, principals can be either code *owners* or code *users*. Code owners are associated with source programs statically, usually in an unforgeable manner, and are understood as the identity of the source, or producer, of the code. Code users, on the other hand, are program consumers that are assigned dynamically to programs, e.g. a UNIX user. Here we will interpret principals as code owners, since this is standard for PL-based security, and since it will be feasible to treat this interpretation statically, with types.

The most widely used PL-based access control system is the JDK security architecture of Java [10]. Among other applications, the architecture is used to support sandboxing of applets, which are prevented from accessing any local files and can communicate via `http` only with their source `url`. However, this is just an application of the general security mechanism, which allows definition of more sophisticated security policies than the simple and restrictive one expressed in sandboxing. We will consider the general JDK1.2 security model, including the *stack inspection* algorithm at the heart of the implementation.

Stack inspection allows for an enforcement of ACL security that prevents untrusted code from sneakily interposing itself into trusted operations. For example, suppose *print*ing is a trusted resource locally. The local system could provide a function *safePrint* for general use, which checks that the caller is authorized for printing before doing it. If code owned by a principal unauthorized for this privilege tries to use *safePrint*, the check will fail. In fact, stack inspection goes further; by literally examining the call stack frames, which are annotated with the identities of the owners of the associated code, and thus the identities of possible users of the active function evaluation, the algorithm ensures that no untrusted code gains even indirect access to the resource, e.g. through man-in-the-middle attacks.

While stack inspection-style access control is a sound model that has been useful in practice, it has some shortcomings, including its effect on run-time efficiency, and the clarity of security policies. We will explore ways to increase the efficiency of stack inspection, and define concise and declarative type terms that improve the clarity of security specifications. In this the current thesis is similar to and inspired by [45], where a compile-time program transformation is described for optimization of stack inspection, and BAN-type logics are used to characterize the system— though here we advocate static analyses, rather than compiler transformations and BAN logics.

In the object confinement security model, access to resources is obtained by gaining possession of a *capability* to the resource, which is an unforgeable reference to the resource along with possibly other information, e.g. an interface. Enforcement of security is based on distribution of these references. For example, Java provides a basic object confinement scheme: by disallowing pointer arithmetic and buffer overflow, the language ensures that the only way to gain access to an object is by being explicitly given a reference. In fact, in the above example of *safePrint*, there is an inherent confinement, in that code in general can obtain a reference to *safePrint*, but not to the primitive *print* function that the system must call to do the actual printing— otherwise, the entire system would be circumvented.

A principal appeal of the object confinement model is its simplicity; since there are no ACLs or run-time stack inspections to perform, the system is very efficient. However, object protection schemes may be more sophisticated, e.g. the Secure Network Objects of [42] that use a distributed capability architecture to implement secure, remote object message sends. In this thesis we will develop an object confinement model that is general enough to capture a wide variety of schemes. All object confinement mechanisms owe a great deal to previous research in capability-based operating systems, including [12, 35, 48]. Significant work has been done to develop systems-level security of this sort, including a formal verification of security properties in [36] that is akin to work here, insofar as that presentation provides formal, foundational results for resource protection.

While there are differences between access control and object confinement, with various benefits and payoffs, the *safePrint* example above suggests that the models can interact. In fact, a large body of recent work in PL-based security [2, 9, 20, 21] has demonstrated a need for *both* the access control and object confinement models in OO languages such as Java. Significantly, as some of these presentations point out, confinement security is needed to prevent ACLs themselves from leaking and being unsafely manipulated, which would completely subvert access control security. Thus, we do not argue that one of these models is better than the other. Rather, we recognize that both are useful, and show how both may benefit from static analyses, thus demonstrating the general applicability of types to secure PLs.

Type Systems for PL-Based Security

Type systems offer various benefits that have been demonstrated in practice, by the success of typed languages such as ML, OCaml and Java. These include *declarative* benefits, since types themselves serve as concise, readable specifications of run-time behavior. They also include *safety* benefits, since type systems rule out programs that exhibit unsafe run-time behavior. Recent work has also demonstrated that types can *optimize* the run-time performance of programs, since compilers can generate better code if more is known about program properties, via types, at compile time.

In this thesis, we show how these benefits may be applied to security mechanisms in PLs, by the use of types that statically analyze security information. In the context of secure programming, run-time safety directly translates to system security, and so is of particular concern. Furthermore, security bugs often originate from a misunderstanding of how the security framework is properly used, not from fundamental flaws in the framework itself. Specifications that are difficult to read are easy to get wrong; thus, the usual declarative benefits of static type frameworks provide a particular advantage over purely dynamic approaches to the enforcement of security. Dynamic security mechanisms can also introduce complications and inefficiencies to language implementations, so any run-time optimizations that can be gained through static analyses are of significant interest.

Since the application is natural, type systems for PL-based security are currently a popular research topic. Many different static approaches to PL-based security have been proposed, perhaps not directly applicable to access control or object confinement. These include type systems for enforcing security in the information flow model, in sequential languages [15, 27] as well as process calculi such as the π-calculus [16, 26]. The proof-carrying-code (PCC) framework [1, 22, 34] is an extremely powerful and expressive framework for the static verification of program security properties, where assertions made about programs in extensible systems are automatically verified. The PCC system is actually more general than type systems; assertions may be so complex as to require human composition. Other novel type systems for PL security have been developed, including static analyses for enforcing safe resource consumption [6], and verifying behavior of expressive "security automata" [44].

Most directly related to work in this thesis are previous static approaches to object confinement [2, 3]. These treat systems that are very similar, though less general than the system treated in Chapter 5; also, our system implements security checks in a distinct manner. The results in Chapters 1 and 4 are versions of material originally presented by the current author, among others, in [28, 37], which represent the first type systems dealing expressly with stack-inspection-based access control. However, approaches to this same issue have since been developed by others [21].

The material presented in this thesis is distinguished from previous work in a number of ways. There are various technical distinctions that will be discussed at greater length in later Chapters. More generally, the principal contributions consist of expressive, flexible type systems that are easily readable, simple, and efficient, with rigorous mathematical foundations. The type systems for both the access control and object confinement model will be designed to reflect the nature of the security context. The type analysis for the access control model will specify the privileges required to perform an action. The type analysis for the object confinement model will specify the domains to which an object is confined. Since these types naturally communicate the security policies of programs, they serve as a clarification of the policies to the programmer. These type systems, though expressive in distinct ways, will both be based on polymorphic *row* types [29], which are concise yet powerful, as well as well-studied and characterized. This significantly raises confidence in the formal rigor of the systems.

Furthermore, these analyses will be *inferrable*; that is, type inference will be an available implementation technique. This is especially important with respect to Java, since it means the analysis could be adopted to the existing Java code base, without a retrofit of type annotations as required by type checking. Furthermore, since our analyses will be based on row types, we will be able to appeal to a large existing codebase for the implementation of the type systems, that includes well-developed methods for efficient type inference [24, 31]. This distinguishes our analysis from e.g. PCC, which is concerned with possibly more complicated properties, and requires the programmer to annotate programs with assertions relevant to its static analysis; our approach ensures that we stay within the bounds of an efficient analysis that imposes no additional overhead on the programmer.

Our analyses promote efficiency by statically enforcing security, allowing dynamic checks to be removed. By proving type soundness results that treat run-time security behavior of programs, we rigorously establish that well-typed programs are dynamically secure, meaning that costly run-time checks such as stack inspection can be eliminated. In fact, our type analysis for object confinement is sensitive enough so that virtually no dynamic checks are needed. This contributes to the feasibility of secure PLs for widespread application, by ensuring their efficiency.

In general, this thesis will pay particular attention to developing readable type systems, to the verification of their correctness via rigorous proof of type safety properties, and to a consideration of how types can improve program performance. While the results are foundational, the applications are practical, insofar as the usefulness of types has been demonstrated, and the ideas presented here can be implemented in an efficient manner. As alluded to above, this last fact is ensured by the re-use of well-studied methods that have already been proven efficient in practice, which is allowed by our translational methodology, discussed in the next section.

The Translational Approach and HM(X)

Developing type systems for novel, non-trivial languages is an involved task. Designing a concise, readable type language is difficult, as is proving the system correct. In this thesis, we will endow languages with an *operational* semantics, which usually implies that a *subject reduction* result is required to prove type safety. Subject reduction is notoriously tedious to prove; however, our methodology will allow *re-use* of an existing subject reduction result, eliminating the most time-consuming hurdle in our syntactic type safety proofs. It will also allow re-use of the elegant *row type* system of [29], including inference methods, resulting in safe, concise, and readable types, which are pre-equipped with efficient implementations.

One aspect of our methodology is the transformational approach to type system development, developed and used previously in [27] and [33]. This approach is characterized by a translation from a novel *source* language, with no static analysis defined, into a previously studied *target* language that comes pre-equipped with a sound type system. By verifying that the source-to-target translation preserves program semantics, a sound *indirect* static analysis for the source language is immediately obtained as a corollary of type soundness in the target language. We may also base a *direct* type analysis for the source language, which treats source expressions directly rather than via transformation, on the program transformation and target type system. This approach allows significant insight into the best form of the source language type system, and saves considerable effort in its soundness proof; rather than proving soundness *ab initio*, a significant task due to the requirements of subject reduction, only a trivial correspondence between the direct and indirect type system need be demonstrated.

We will use the transformational approach to develop type systems for both our access control and object confinement models. In fact, we will be able use the same target language in each case: it is interesting to note that the same language can be used to capture the behavior of such distinct security models. Our target language, called pml_B, is based on Rémy's Projective ML [30], and is similarly endowed with row types [29]; however, it contains significant novelties, including language features and concise, accurate types for operations on sets of atomic elements such as intersection, union and difference. To define these types, we use *conditional constraints* [24]. This language and type system will be developed by instantiating the HM(X) framework [23, 40], which allows re-use of a stateful functional core and type system, including subject reduction and type soundness results. This approach significantly simplifies a proof of type safety for pml_B, since, again, it is not necessary to prove subject reduction *ab initio*.

An ancillary result of this thesis is subject reduction for HM(X), which has not been previously demonstrated; we prove it here mainly to fill a gap in the literature (in fact, a soundness proof for HM(X) with respect to a denotational semantics exists in [40], and a so-called *semi-syntactic* result is proved in [25]; the latter would be sufficient to prove syntactic type soundness for pml_B here). Since the relevant proof requires an entire Chapter, it is easy to see the work that is saved by taking the result as a given.

Structure of the Thesis

The rest of the thesis is structured as follows. The general method of the thesis is presented in Chapter 1 by way of example, where we take a first look at a language with stack inspection security, including operational semantics, type system, type soundness and optimization results. To set the stage for our transformational approach, in Chapter 2 we define the HM(X) system and provide subject reduction and type soundness results. The pml_B language is then presented as an instance of HM(X) in Chapter 3. In Chapter 4 we return to stack inspection security, and develop a more sophisticated language model and type system through transformation into pml_B. We turn our attention to object confinement in Chapter 5, where a language is defined that incorporates this sort of security, as well as a type discipline that enforces it. Again, type soundness and optimization results are discussed, and are obtained via transformation into pml_B. In the Conclusion, we close with some final observations and remarks.

Chapter 1

Types for Access Control: First Look

In this chapter, we examine the stack inspection security mechanism, casting it into a simple model that captures the essential security properties of the Java JDK1.2 stack inspection system. The chapter serves three purposes; to familiarize the reader with stack-inspection security, to describe a low-level model that accurately reflects the description of Java stack-inspection in the literature [10], and to provide an understandable example of the static approach advocated by this thesis, via development of a simple monomorphic type system. Later, in Chapter 5, we will recast our language model into a more technically appealing, but less transparently "JDK1.2-like" form. By proving that this latter form is capable of simulating the form presented in this Chapter, we maintain confidence in the faithfulness to real implementations throughout. In Chapter 5 we will also present a polymorphic type system in full detail, that subsumes the system presented here.

The chapter will proceed as follows. In Sect. 1.1, we briefly describe and discuss the stack inspection model, with observations about its shortcomings. In Sect. 1.2, we provide a language formalization of this model. Then, in Sect. 1.3, we define a static type discipline that resolves shortcomings associated with the current stack inspection implementations.

1.1 Review and critique of Java stack inspection

The stack inspection system described here is a simplified version of that found in Java, intended to capture only the core mechanisms. The reader familiar with the JDK security architecture will note that some of its more complex features, e.g. privilege inheritance and parameterized privileges, are not captured—but our goal here is a solid foundation for static access control, rather than a complete model of the JDK security architecture.

In the JDK, access control lists (ACLs) are defined by owners of the local system, which associate *code owners*, or *principals*, with sets of *privileges*. The stack inspection mechanism is a technique for activating and checking privilege activations. This is fundamentally a *dynamic* security checking system, in that access restrictions are all checked at run-time, not compile-time. To use the system, the programmer adds “do privileged” and “check privilege” commands to the code. A “do privileged” command takes as parameter a “privileged action” object, which must contain a *run* method, which takes no parameters, that is a macro for some sequence of security-sensitive actions. This run method is invoked in the body of a “do privileged” call, and the stack frame associated with the invocation is annotated with a privilege flag. When a privilege is checked via the “check privilege” command, stack frames are searched most to least recent. If a frame is encountered with the desired flag, the check succeeds. Additionally, all programs in JDK 1.2 come with a specified owner, and stack frames are annotated with the name of the owner of the code associated with each frame. If an owner who is unauthorized for the privilege being checked is encountered on a stack frame before success of inspection, the check fails. For more Java-specific detail, see [11, 19] for a concise description of the stack inspection mechanism in Java, and how it may be used to enforce security properties. The reader is referred to [45] for good examples of the use and advantages of the JDK 1.2 architecture. One appealing aspect of the system is that any security policy is highly programmer-specified– the programmer inserts privilege activations and checks where they are appropriate. This makes the system more malleable than the information-flow model [7], for example.

The Java security architecture is a solid proposal which is being applied in practice, but it has significant flaws. There is a performance penalty due to the need for run-time stack inspection; in addition to the presence of run-time security checks, compiler optimizations such as tail-call elimination and CPS transformation would interfere with the stack inspection semantics, and are thus precluded. Our static approach eliminates the need for certain run-time checks, allowing an interaction of stack inspection security and these compiler optimizations. Another solution to this problem is called security-passing style (SPS) [45], which eliminates the need to literally inspect every stack frame. However, even this solution does not address the *ad hoc* nature of the architecture; all security properties are enforced by method calls, a highly non-declarative form of specification. This makes the access control specification very difficult to read—it is all buried in the code and the implicit control flow structure of that code. The static analyses we explore do address this problem.

$x \in ID$	*identifiers*
$r \in \mathcal{R}, R \subseteq \mathcal{R}$	*resources*
$p \in \mathcal{P}, \mathcal{A} \subseteq \mathcal{P} \to 2^{\mathcal{R}}$	*principals & ACL's*
$v ::= \mathtt{C}(\gamma, \lambda x.e)$	*values*
$e ::= x \mid v \mid \lambda x.f \mid e\, e \mid \mathsf{dopriv}_r e \mid \mathsf{check}\, r\, \mathsf{then}\, e \mid \cdot e \cdot \mid f$	*expressions*
$f ::= p.e$	*signed expressions*
$E ::= [] \mid E\, e \mid v\, E \mid \mathsf{dopriv}_r E \mid \cdot E \cdot$	*evaluation contexts*
closed $v \in \mathbb{V}, \gamma \in ID \rightharpoonup \mathbb{V}$	*environments*
$\mathbf{f} ::= \mathbf{off} \mid \mathbf{on}(r)$	*activation flags*
$\mathtt{S} ::= nil \mid \langle \gamma, p, \mathbf{f} \rangle :: \mathtt{S}$	*stacks*

Figure 1.1: Grammar for $\lambda_{\text{sec}}^{\mathsf{S}}$

1.2 The $\lambda_{\text{sec}}^{\mathsf{S}}$ language definition

This section defines $\lambda_{\text{sec}}^{\mathsf{S}}$, a simplified model of the JDK 1.2 security architecture. It is a λ-calculus equipped with a notion of code ownership, access control lists (ACLs), and constructs for activating and checking privileges. A "low-level" operational semantics is defined, comprising a representation of call stacks with security annotations. This form clearly reflects actual implementations.

1.2.1 Syntax

We assume given an arbitrary set $\mathcal{R}$ of *privileges*, using r and R to range over privileges and over sets thereof, respectively. For simplicity we assume a fixed ACL $\mathcal{A}$, although the formalization is adaptable to any local definition of $\mathcal{A}$. Any $\mathcal{A}$ is a map from principals p to sets of privileges R. We distinguish a principal *nobody* to denote an anonymous principal, with $\mathcal{A}(\textit{nobody}) = \varnothing$.

The grammar of $\lambda_{\text{sec}}^{\mathsf{S}}$ is given in Fig. 1.1. A *signed expression* $p.e$ behaves as the expression e endowed with the authority of principal p. The body of every λ-abstraction is required to be a signed expression – thus, every piece of code must be vouched for by some principal; otherwise, although we include f in the language of expressions for technical convenience, "bare" signed expressions $p.e$ are disallowed. The construct dopriv_r allows a principal to activate the use of a resource r within the expression e. The construct $\mathsf{check}\, r\, \mathsf{then}\, e$ asserts that the use of r is activated.

If r is indeed activated, e is evaluated; otherwise, execution fails. We let $\lambda\_.e$ denote the function where $\_$ does not appear free in e.

Since we will be defining a stack-based semantics, $\lambda_{\text{sec}}^{\mathtt{S}}$ also includes a definition of *closures* $\mathtt{C}(\gamma, \lambda x.e)$, which is a function together with a binding environment γ for the free variables in $\lambda x.e$. Binding environments are partial maps from identifiers x to closed values v, denoted $[v_1/x_1, \ldots, v_n/x_n]$. We write $\gamma[x/v]$ to denote the binding environment that is equivalent to γ but which maps x to v, and write $\gamma; \gamma'$ to denote the environment that is equivalent to γ but which maps $x \in \text{dom}(\gamma')$ to $\gamma'(x)$, while $\gamma \backslash x$ denotes the environment which is undefined on x and is otherwise equivalent to γ. We define the free variables of an expression fv(e) as usual, extending the definition to closures as $\text{fv}(\mathtt{C}(\gamma, \lambda x.e)) = \text{fv}(\lambda x.e) - \text{dom}(\gamma)$. An application of a binding to an expression, denoted $e[v_1/x_1, \ldots, v_n/x_n]$, results in the substitution of $v_1, \ldots, v_n$ for free occurrences of $x_1, \ldots, x_n$, respectively, in e. Substitutions are extended to stacks as follows:

$$\begin{array}{rcl} \text{env}(\langle \gamma_1, p_1, \mathbf{f}1 \rangle :: \cdots :: \langle \gamma_n, p_n, \mathbf{f}_n \rangle :: \mathit{nil}) & \triangleq & \gamma_n; \cdots; \gamma_1 \\ \mathtt{S}(e) & \triangleq & e(\text{env}(\mathtt{S})) \end{array}$$

For the purposes of our stack-based semantics, we also have *framed* expressions $\cdot e \cdot$, denoting a region of code associated with a stack frame. These expressions, and closures as well, are for the purposes of operational bookkeeping, and we say that e is a *top-level* expression iff e is closed and contains no subexpressions of the form $\cdot e' \cdot$ or $\mathtt{C}(\gamma, e')$. Additionally, we disallow λ-abstractions and closures with framed subexpressions.

1.2.2 The stack inspection algorithm

We now give the definition of stack inspection in our model, which is a formalization of the description in Sect. 1.1.

The $\lambda_{\text{sec}}^{\mathtt{S}}$ grammar contains a definition of stacks $\mathtt{S}$. Each $\mathtt{S}$ is a stack of frames $\langle \gamma, p, \mathbf{f} \rangle$, where γ is a binding environment, p is the owner identity associated with the frame, and $\mathbf{f}$ is a privilege activation annotation. Each frame is an activation record associated with a function call, with γ the free-variable bindings for the function, p the function code owner, and $\mathbf{f}$ denoting whether the function was executed normally, in which case $\mathbf{f} = \mathbf{off}$, or as the result of a call to dopriv_r, in which case $\mathbf{f} = \mathbf{on}(r)$. The inspect function inspects the stack for an activation of a privilege r, in

$$
\begin{array}{rcllr}
\mathtt{S}, x & \to & \mathtt{S}, \mathtt{S}(x) & & (\textit{var}) \\
\mathtt{S}, \lambda x.e & \to & \mathtt{S}, \mathtt{C}(\mathrm{env}(\mathtt{S}), \lambda x.e) & & (\textit{closure}) \\
\mathtt{S}, \mathtt{C}(\gamma, \lambda x.p.e)v & \to & \langle \gamma[v/x], p, \mathbf{off} \rangle :: \mathtt{S}, \cdot e \cdot & & (\textit{app}) \\
\mathtt{S}, \mathsf{dopriv}_r\, \mathtt{C}(\gamma, \lambda\_.p.e) & \to & \langle \gamma, p, \mathbf{on}(r) \rangle :: \mathtt{S}, \cdot e \cdot & & (\textit{dopriv}) \\
\mathtt{S}, \mathsf{check}\, r\, \mathsf{then}\, e & \to & \mathtt{S}, e & \text{if } \mathrm{inspect}(\mathtt{S}, r) = \textbf{true} & (\textit{checkpriv}) \\
\langle \gamma, p, \mathbf{f} \rangle :: \mathtt{S}, \cdot v \cdot & \to & \mathtt{S}, v & & (\textit{pop}) \\
\mathtt{S}, E[e] & \to & \mathtt{S}', E[e'] & \text{if } \mathtt{S}, e \to \mathtt{S}', e' & (\textit{context})
\end{array}
$$

Figure 1.2: Operational semantics of $\lambda_{\text{sec}}^{\mathtt{S}}$

the manner employed in Java.

$$
\begin{array}{rcll}
\mathrm{inspect}(\mathit{nil}, r) & = & \textbf{false} & \\
\mathrm{inspect}(\langle \gamma, p, \mathbf{on}(r) \rangle :: \mathtt{S}, r) & = & \text{if } r \notin \mathcal{A}(p) \text{ then } \textbf{false} \text{ else } \textbf{true} & \\
\mathrm{inspect}(\langle \gamma, p, \mathbf{on}(r') \rangle :: \mathtt{S}, r) & = & \text{if } r \notin \mathcal{A}(p) \text{ then } \textbf{false} \text{ else } \mathrm{inspect}(\mathtt{S}, r) & \text{where } r' \neq r \\
\mathrm{inspect}(\langle \gamma, p, \mathbf{off} \rangle :: \mathtt{S}, r) & = & \text{if } r \notin \mathcal{A}(p) \text{ then } \textbf{false} \text{ else } \mathrm{inspect}(\mathtt{S}, r) &
\end{array}
$$

This algorithm implements Java stack inspection: given a privilege r, the stack is searched frame by frame from the current frame until the privilege is found on the stack (return **true**), or the owner of the frame lacks that credential (return **false**), or we ran off the top of the stack (return **false**). The set of privileges which are enabled on a particular stack $\mathtt{S}$, given some access credential list $\mathcal{A}$, is denoted $\mathrm{privs}(\mathtt{S})$; *i.e.*,

$$\mathrm{privs}(\mathtt{S}) = \{r \mid \mathrm{inspect}(\mathtt{S}, r) = \textbf{true}\}$$

1.2.3 Operational semantics

With the syntax of $\lambda_{\text{sec}}^{\mathtt{S}}$ and stack inspection defined, we may now give the operational semantics, defined in Fig. 1.2, which is a reduction relation $\to$ on *configurations* $\mathtt{S}, e$. We specify that $\to$ be defined only on *well-formed* configurations, which we elaborate as follows:

Definition 1.1 *The* frame depth *of an evaluation context is inductively defined as follows: the frame depth of* $[\,]$ *is* 0*, the frame depth of* $\cdot E \cdot$ *is* 1 *plus the frame depth of* E*, and the frame depth of any other context form* E *is the frame depth of* E*'s subcontext.*

Definition 1.2 *A configuration δ, e, σ is* well-formed with respect to $\rightarrow$ *iff $\sigma(e)$ is closed, and there exists E and unframed e' such that $e = E[e']$ and the frame depth of E equals the length of δ, where the* length *of a stack $(\langle \gamma_1, p_1, \mathbf{f}1 \rangle :: \cdots :: \langle \gamma_n, p_n, \mathbf{f}_n \rangle :: nil)$ is n..*

Several of these rules implement a semantics whereby variable bindings are kept on the stack and looked up when necessary. This includes the rule for functions $\lambda x.e$, which reduce to closures $\mathtt{C}(\gamma, \lambda x.e)$. Note that in a normal function call, the new stack frame is annotated with **off**, whereas if dopriv_r is applied to a function closure, then a new frame associated with an application of the closure is annotated with $\mathbf{on}(r)$– note that the closure must contain a function that takes no parameters, as must the run method of privileged action objects in Java.

We let $\rightarrow^\star$ denote the reflexive, transitive closure of $\rightarrow$. We say that $\mathtt{S}, e$ is *stuck* if e is not a value but there is no $\mathtt{S}', e'$ such that $\mathtt{S}, e \rightarrow \mathtt{S}', e'$— that is, stuck $\mathtt{S}, e$ are semantically meaningless. If $nil, e \rightarrow \mathtt{S}', e'$ and $\mathtt{S}', e'$ is stuck, then we say that e *goes wrong*.

1.3 Types for $\lambda_{\text{sec}}^{\mathsf{S}}$

The previous section specified the language and dynamic behavior of $\lambda_{\text{sec}}^{\mathsf{S}}$. In this section, we propose a static analysis for $\lambda_{\text{sec}}^{\mathsf{S}}$ that allows the declaration and enforcement of security properties of programs. This analysis allows runtime checks to be eliminated during the execution of $\lambda_{\text{sec}}^{\mathsf{S}}$ programs.

The idea behind this type system, is that privilege *needs* of a program is represented in its type, where needs are those privileges that are checked during execution of the program. Functions f that perform privileged actions will have type $\tau \xrightarrow{\varsigma} \tau'$, containing a needs annotation ς. The type system statically verifies that needs of expressions would met by dopriv_rs during execution. These "security types" ς are actually *set types*, types which can accurately describe the contents of sets of urelements. In this case, the urelements are privileges. Thus, if a function has type $\tau \xrightarrow{\varsigma} \tau'$ and requires r_1 and r_2 to be activated for its use, then ς is the type of the set $\{r_1, r_2\}$. However, we note that these sets are *implied* by the semantics of the language; sets are no way an actual language construct of $\lambda_{\text{sec}}^{\mathsf{S}}$.

1.3.1 Type language

The language of types τ includes monomorphic type variables, function types, and set types. Set types are composed of elements rc, where c is either $+$, denoting that the element is

$$
\begin{array}{rcll}
\tau & ::= & \alpha, \beta, \ldots \mid \tau \to \tau \mid \{\tau\} \mid \varnothing \mid b\tau, \tau \mid c & \textit{types} \\
c & ::= & + \mid - & \textit{constructors}
\end{array}
$$

Figure 1.3: Type and constraint grammar for $\lambda_{\text{sec}}^{\mathsf{S}}$

$$
\frac{\alpha \in \mathcal{V}_k}{\alpha : k} \qquad \frac{\tau, \tau' : \mathit{Type} \qquad \tau'' : \mathit{Set}_{\varnothing}}{\tau \xrightarrow{\{\tau''\}} \tau' : \mathit{Type}} \qquad \varnothing : \mathit{Set}_B \qquad c : \mathit{Con}
$$

$$
\frac{\tau : \mathit{Con} \qquad b \notin B \qquad \tau' : \mathit{Set}_{B \cup \{b\}}}{(b\tau, \tau') : \mathit{Set}_B}
$$

Figure 1.4: Type kinding rules for $\lambda_{\text{sec}}^{\mathsf{S}}$

present in the set, or $-$, denoting that element is absent. For technical reasons that will become clear throughout this thesis, as well as for coming language extensions, the flexibility of specifying the presence or absence of a particular element is essential.

Set types also contain constructors $\varnothing$ and ω, denoting whether all other elements not explicitly mentioned in the rest of the type are present or absent, respectively. For example, if a function has type $\tau \xrightarrow{\varsigma} \tau'$ and requires r_1 and r_2 to be activated for its use, then ς has type $\{r_1+, r_2+, \varnothing\}$. Note that this informal description of the constructors $\varnothing$ and ω implies certain equational properties of set types, e.g.:

$$
\{r_1+, r_2+, \varnothing\} = \{r_1+, r_2+, r_3-, \varnothing\} = \{r_1+, r_2+, r_3-, r_4-, \varnothing\} = \ldots
$$

We delay a formal account of this behavior until Chapter 3.

To ensure that only meaningful types can be built from the type grammar, we equip types with *kinds* in Fig. 1.4; from here on, we assume that any type is *well-kinded*, in the sense of obeying the kinding rules. Note that these rules ensure that set types do not have repeated elements. Formally, we let ρ range over types of kind $\mathit{Set}_{\varnothing}$, and let ς range over types of the form $\{\rho\}$.

1.3.2 Type judgements

Type judgements in $\lambda_{\text{sec}}^{\mathsf{S}}$ are of the form $p, \varsigma, \Gamma \vdash e : \tau$, where p represents the code owner, ς represents the set of currently active privileges, and Γ is a type binding environment. We write $(\Gamma; x : \tau)$ to denote the environment which binds x to τ, and which otherwise is equivalent to Γ.

The type judgement rules are then given in Fig. 1.5. The most novel rules are those associated with dopriv_r, check, and signed expressions. The rule SIGN for signed expressions $p.e$ ensures that any privileges needed to execute e are in $\mathcal{A}(p)$— that is, are authorized to p. The rule CHECK ensures that a privilege being checked is in the currently active set. The rule DOPRIV SUCCESS activates the specified privilege r in its precedent; DOPRIV FAILURE has no effect, in case the specified r is not authorized to the current principal. In derivations we disallow the judgement $p, \varsigma, \Gamma \vdash x : \tau_{dp}$ in any consequence of VAR; this statically enforces that any argument of dopriv reduces to a closure with a dummy argument. We say a judgement is *valid* iff it can be derived according to these rules, and we say e is *well-typed* iff $nobody, \{\varnothing\}, \varnothing \vdash e : \tau$ is valid, in which case we write $e : \tau$.

1.3.3 Type safety

One of the main points of this formalization is to provide a foundation for rigorously proving *type safety*— that is, for proving that only semantically meaningful expressions are typable. In the case of $\lambda_{\text{sec}}^{\mathsf{S}}$, this implies that only *secure* expressions are typable, since the semantics ensures that insecure computations fail. By establishing type safety, we also prove that run-time security checks can be eliminated, since well-typedness ensures that all such checks will succeed. Type safety comprises both *soundness* and *progress* results, which we may state here as follows:

Theorem 1.1 ($\lambda_{\text{sec}}^{\mathsf{S}}$ Type Safety) *If top-level e is well-typed then e does not go wrong.*

Theorem 1.2 ($\lambda_{\text{sec}}^{\mathsf{S}}$ Progress) *If top-level e is well-typed then either $nil, e \to^{\star} \mathtt{S}, v$ or e diverges.*

One important consequence of these results is that we may now formally assert that runtime stack inspection is no longer necessary:

Proposition 1.1 *Let $\leadsto$ be defined as $\to$, but with calls to* inspect *eliminated. Suppose e is well-typed; then $nil, e \leadsto^{\star} \mathtt{S}', v$ iff $\mathtt{S}, e \to^{\star} \mathtt{S}, v$.*

At this point it would be traditional to develop *ab initio* proofs of the correctness of these assertions. However, we will instead lay aside these proofs, returning to them in Chapter 3 when we

$$
\frac{\Gamma(x) = \tau}{p, \varsigma, \Gamma \vdash x : \tau}\ \textsc{Var}
\qquad
\frac{\star, \varsigma_2, (\Gamma; x : \tau_1) \vdash f : \tau_2}{p, \varsigma_1, \Gamma \vdash \lambda x.f : \tau_1 \xrightarrow{\varsigma_2} \tau_2}\ \textsc{Abs}
\qquad
\frac{p, \varsigma, \Gamma \vdash e_1 : \tau_2 \xrightarrow{\varsigma} \tau \qquad p, \varsigma, \Gamma \vdash e_2 : \tau_2}{p, \varsigma, \Gamma \vdash e_1\, e_2 : \tau}\ \textsc{App}
$$

$$
\frac{p, \{\rho\}, \Gamma \vdash e : \tau_{dp} \xrightarrow{\{\rho\}} \tau \qquad r \notin \mathcal{A}(p)}{p, \{\rho\}, \Gamma \vdash \mathsf{dopriv}_r e : \tau}\ \textsc{Dopriv Failure}
\qquad
\frac{p, \{r+, \rho\}, \Gamma \vdash e : \tau_{dp} \xrightarrow{\{r+,\rho\}} \tau \qquad r \in \mathcal{A}(p)}{p, \{r\varphi, \rho\}, \Gamma \vdash \mathsf{dopriv}_r e : \tau}\ \textsc{Dopriv Success}
$$

$$
\frac{p, \{r+, \rho\}, \Gamma \vdash e : \tau}{p, \{r+, \rho\}, \Gamma \vdash \mathsf{check}\, r\, \mathsf{then}\, e : \tau}\ \textsc{Check}
$$

$$
\frac{p, \{r_1\varphi_1, \ldots, r_n\varphi_n, \varnothing\}, \Gamma \vdash e : \tau \qquad \mathcal{A}(p) = \{r_1, \ldots, r_n\}}{\star, \{r_1\varphi_1, \ldots, r_n\varphi_n, \rho\}, \Gamma \vdash p.e : \tau}\ \textsc{Sign}
$$

Figure 1.5: Type judgement rules for $\lambda_{\text{sec}}^{\mathsf{S}}$

develop a polymorphic type system that subsumes the current one, including its safety properties. A central point of this thesis is that proofs of properties such as these, which take a great deal of effort with an *ab initio* approach, can be made much easier via the transformational approach we will use to develop the polymorphic type system for $\lambda_{\text{sec}}^{\mathsf{S}}$.

1.4 $\lambda_{\text{sec}}^{\mathsf{S}}$ language and type examples

Here we give some examples that demonstrate the use of $\lambda_{\text{sec}}^{\mathsf{S}}$, the readability of types for the language, and how types are used to enforce security.

Assume the following definitions:

$$
\begin{aligned}
\mathit{ok} &\triangleq \lambda x.p.x \\
\mathit{check}_r &\triangleq \lambda\_.p.\mathsf{check}\, r\, \mathsf{then}\, \mathit{ok}
\end{aligned}
$$

To these expressions we may assign the following types, where τ_{ok} refers to the specified type of

τ_{ok} hereafter:

$$\begin{array}{rcl} ok & : & \alpha \xrightarrow{\{\beta\}} \alpha \\ check_r & : & \tau_{dp} \xrightarrow{\{r+,\beta\}} \tau_{ok} \end{array}$$

Note that the security needs of $check_r$ show up explicitly in its type, whereas the type of ok reflects no security requirements. Now, at the top level with no privileges enabled, the expression $check_r\, ok$ is operationally unsafe, since $check_r$ requires that r be enabled to be applied. Appropriately, the expression $check_r\, ok$ is not well-typed, since typing the expression with the ABS rule requires that $check_r$ has no statically determined top-level needs.

Extending our examples, we consider "boilerplate" functions that a system might provide for users to safely perform privileged actions. In particular, a function that enables r for a privileged action is defined as follows:

$$enable_r \triangleq \lambda f.p.(\mathsf{dopriv}_r\, f)$$

Assuming $\mathcal{A}$ such that $r \in \mathcal{A}(p)$, this function may be given the following type:

$$enable_r : (\tau_{dp} \xrightarrow{\{r+,\beta\}} \tau_{ok}) \xrightarrow{\{\beta'\}} \tau_{ok}$$

This type specifies that $enable_r$ may be applied to a function f that performs an action requiring that r be enabled ($r+$), returning the result of this action without the requirement that r be enabled, so that:

$$enable_r check_r \quad : \quad \tau_{ok}$$

Thus, the security of programs is statically verified.

1.5 Looking forward

In this chapter, we have defined an initial, intuitively correct language model for stack inspection security, called $\lambda^{\mathsf{S}}_{\mathrm{sec}}$. We have also developed a simple monomorphic type system for $\lambda^{\mathsf{S}}_{\mathrm{sec}}$, but have postponed proving type safety until a more sophisticated, polymorphic system is developed. As discussed in the introduction, our development of a polymorphic type system for $\lambda^{\mathsf{S}}_{\mathrm{sec}}$ is accomplished via transformation into another language; the next two chapters will establish the preliminary results for this transformation. In Chapter 4, we will return to $\lambda^{\mathsf{S}}_{\mathrm{sec}}$, extending and recasting the language into a more technically appealing form— in particular, explicit stacks will be eliminated— while demonstrating that the new language subsumes the language presented in this chapter. We will then equip the language with a polymorphic type system that is proven safe in a manner that is easier than the traditional *ab initio* approach; the proofs of Theorem 1.1, Theorem 1.2 and Proposition 1.1 will fall out as corollaries of these results.

Chapter 2

Technical Interlude: the System HM(X)

In this Chapter the HM(X) framework is described, including definitions of the language, constraint and type systems. Type inference is also defined in Sect. 2.4, including relevant correctness results. An OCaml implementation of type inference, included in the appendix, is described and discussed.

We also present the first purely syntactic type soundness results for HM(X), in the style of [47]. A soundness result based on a denotational semantics is presented in [23], and a *semi-syntactic* result, based on an interpretation of HM(X) type judgements in an intermediate system, is presented in [25]. The former result is inadequate in the event that a syntactic result is desired for some instance of the framework; the latter is more satisfactory in this sense, but lacks subject reduction. The purely syntactic results presented here, including subject reduction, thus serve as a direct verification of HM(X) type soundness with respect to its operational semantics.

This presentation of HM(X) extends previous results by treating a version of the core language that contains state and a primitive recursive binding mechanism. The addition of state increases the expressivity of the programming language. A primitive recursive binding mechanism is a welcome convenience; previously, it was necessary to either define a fixpoint combinator, or introduce one as a constant, entailing additional proof overhead to obtain type soundness for an instance of the framework.

Our presentation of HM(X) is otherwise identical to that of [23, 40]. The main difference is our interpretation of constraints, which is more direct; see section 2.1.2. Our proof technique is standard, following Wright and Felleisen [47]. The central results are subject reduction, progress, and type safety for the HM(X) framework, stated and proved in section 2.3.

$$\begin{array}{rcll}
x, z & \in & \mathit{ID} & \textit{identifiers} \\
l & \in & \mathit{Loc} & \textit{memory locations} \\
\mathbf{c} & \in & \mathit{Const} & \textit{constants} \\
v & ::= & x \mid l \mid \mathsf{fix}\, z.\lambda x.e \mid \mathsf{ref} \mid := \mid (:= l) \mid ! \mid \mathbf{c} & \textit{values} \\
e & ::= & v \mid e\, e \mid \mathsf{let}\, x = v \,\mathsf{in}\, e & \textit{expressions} \\
E & ::= & [\,] \mid E\, e \mid v\, E & \textit{evaluation contexts}
\end{array}$$

Figure 2.1: Language grammar for HM(X)

2.1 Definitions

In this section we present the HM(X) framework, that is, the programming language and its type system.

2.1.1 The Language

The core language is a call-by-value functional calculus, extended with a recursive binding mechanism built into function definitions, and mechanisms for state. We postulate countably infinite sets of identifiers, locations, and constants. The language grammar is defined in figure 2.1. Note that, following [46], we impose a *value restriction* on let bindings, precluding unsafe interaction between imperative features and polymorphism; for convenience, we define the syntactic sugar $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \triangleq (\lambda x.e_2)e_1$ in case e_1 is not a value.

The operational semantics is defined on *configurations* e, σ, where a *store* σ is a partial mapping from locations to values. We write $\sigma[l \mapsto v]$ to denote the store which maps l to v and otherwise agrees with σ. The empty store is denoted $\varnothing$. The one-step reduction rules for HM(X) are then defined in figure 2.2. We write $\to^\star$ to denote the reflexive, transitive closure of $\to$. The interpretation of constants is given by a (possibly partial) function δ which maps a pair of a constant and a closed value to a closed value.

2.1.2 Constraint Systems

Any instance of the HM(X) framework is parameterized by a *constraint system*. This system must at least comprise the following language of types and constraints, where $\mathcal{V}$ is a countably

$$
\begin{array}{rcllr}
(\mathsf{fix}\, z.\lambda x.e)v, \sigma & \to & e[v/x][\mathsf{fix}\, z.\lambda x.e/z], \sigma & & (\beta) \\
\mathsf{let}\, x = v\, \mathsf{in}\, e, \sigma & \to & e[v/x], \sigma & & (\mathit{let}) \\
\mathrm{ref}\, v, \sigma & \to & l, \sigma[l \mapsto v] & l \notin \mathrm{dom}(\sigma) & (\mathit{ref}) \\
:= l\, v, \sigma & \to & v, \sigma[l \mapsto v] & l \in \mathrm{dom}(\sigma) & (\mathit{assign}) \\
!l, \sigma & \to & \sigma(l), \sigma & & (\mathit{deref}) \\
\mathbf{c}\, v, \sigma & \to & \delta(\mathbf{c}, v), \sigma & & (\delta) \\
E[e], \sigma & \to & E[e'], \sigma' & \text{where } e, \sigma \to e', \sigma' & (\mathit{context})
\end{array}
$$

Figure 2.2: Operational semantics for HM(X)

infinite set of type variables:

$$
\begin{array}{lll}
\alpha, \beta \in \mathcal{V} & & \textit{type variables} \\
\tau \;\; ::= & \alpha \mid \tau \to \tau \mid \tau \text{ ref} \mid \ldots & \textit{types} \\
C \;\; ::= & \mathbf{true} \mid \tau = \tau \mid \tau \leq \tau \mid C \wedge C \mid \exists \alpha.C \mid \ldots & \textit{constraints}
\end{array}
$$

To interpret constraints, we adopt the model-based approach described in [25], which is established via a mapping from types into a universe of partially ordered monotypes T.

Definition 2.1 (Model) *Let $(T, \leq)$ be a partially ordered set, where $t \in T$ is called a* monotype*. Let $\to$ be a function from $T \times T$ into T, where $t_1 \to t_2 \leq t_1' \to t_2'$ implies $t_1' \leq t_1$ and $t_2 \leq t_2'$. Let* ref *be a function from T to T, such that t* ref *$\leq t'$* ref *implies $t = t'$. We require t_1* ref *$\leq t_2 \to t_3$ and $t_2 \to t_3 \leq t_1$* ref *to be false for any $t_1, t_2, t_3 \in T$.*

Definition 2.2 (Interpretation) *An* assignment *ρ is a total mapping from $\mathcal{V}$ to T. An* interpretation *of a constraint system consists of an extension of assignments to arbitrary types, and a* constraint satisfaction relation*, denoted $\rho \vdash C$. The interpretation is* standard *iff the following conditions are satisfied:*

$$
\begin{array}{rcl}
\rho(\tau_1 \to \tau_2) & = & \rho(\tau_1) \to \rho(\tau_2) \\
\rho(\tau \text{ ref}) & = & \rho(\tau) \text{ ref}
\end{array}
$$

$$
\rho \vdash \mathbf{true}
$$

$$
\begin{array}{rcl}
\rho \vdash \tau_1 = \tau_2 & \Leftrightarrow & \rho(\tau_1) = \rho(\tau_2) \\
\rho \vdash \tau_1 \leq \tau_2 & \Leftrightarrow & \rho(\tau_1) \leq \rho(\tau_2) \\
\rho \vdash C_1 \wedge C_2 & \Leftrightarrow & (\rho \vdash C_1) \wedge (\rho \vdash C_2) \\
\rho \vdash \exists \alpha.C & \Leftrightarrow & \exists t.\rho[\alpha \mapsto t] \vdash C
\end{array}
$$

If $\rho \vdash C$ holds, we say that ρ satisfies *or is a* solution *of C. We write $C \Vdash C'$ iff every solution of C is also a solution of C'.*

We identify constraints modulo logical equivalence, that is, we identify C and D when $C \Vdash D$ and $D \Vdash C$ hold. A variable α is deemed *free* in a constraint C iff $C \neq \exists\alpha.C$. We write $\mathrm{fv}(C)$ for the set of all variables free in C.

Our presentation differs from that of Odersky *et al.* [23] by viewing constraints as formulae interpreted in T, rather than as elements of an abstract *cylindric constraint system*. Our presentation is thus perhaps slightly less general, but more concise. Also, we abandon Odersky *et al.ś* notion of constraints in *solved form*. Instead, we identify constraints modulo logical equivalence, which means that we do not care about their syntactic representation. We believe that the representation of constraints is an important issue when designing a constraint solver, but is irrelevant when proving the type system correct.

2.1.3 The Type System

The $\mathrm{HM}(X)$ type system is defined as a system of deduction rules, given in figure 2.3, whose consequents are *judgements* of the form $C, \Gamma \vdash e : \sigma$ where C is a constraint, Γ is a *type environment*, and σ is a *type scheme*. These notions are introduced in the following definition:

Definition 2.3 Type schemes *are of the form $\forall\bar{\alpha}[C].\tau$. Abusing notation, we abbreviate $\forall\varnothing[$***true***$].\tau$ as τ, and abbreviate $\forall\bar{\alpha}[$***true***$].\tau$ as $\forall\bar{\alpha}.\tau$. We identify type schemes modulo α-equivalence.* Type environments Γ *are sequences of bindings of the form $x : \sigma$ and $l : \tau$.*

A type scheme σ is *consistent* with respect to a constraint C if C guarantees that σ has at least one instance. This notion, defined below, appears as a technical side-condition in rule VAR. This extra side-condition is our only deviation from the rules given in [23, 40]. Its effect is to allow some theorems to be stated without a "consistency" requirement on Γ.

Definition 2.4 *We say that a type scheme $\sigma = \forall\bar{\alpha}[D].\tau$ is consistent with respect to a constraint C, and we write $C \Vdash \sigma$, iff $C \Vdash \exists\bar{\alpha}.D$. We say that σ is consistent iff* **true** $\Vdash \sigma$.

Let Δ be a fixed total mapping from the constants to closed, consistent type schemes. Δ is looked up in rule CONST to associate a type scheme with a constant.

Definition 2.5 *A judgement $C, \Gamma \vdash e : \sigma$ is* valid *(or* holds*) iff it is derivable according to the rules of figure 2.3 and C is satisfiable. Then, e is* well-typed.

$$\frac{\Gamma(x) = \sigma \qquad C \Vdash \sigma}{C, \Gamma \vdash x : \sigma}\ \textsc{Var} \qquad \frac{\Gamma(l) = \tau}{C, \Gamma \vdash l : \tau\ \mathsf{ref}}\ \textsc{Loc} \qquad \frac{\Delta(\mathbf{c}) = \sigma}{C, \Gamma \vdash \mathbf{c} : \sigma}\ \textsc{Const}$$

$$\frac{C, (\Gamma; x : \tau; z : \tau \to \tau') \vdash e : \tau'}{C, \Gamma \vdash \mathsf{fix}\, z.\lambda x.e : \tau \to \tau'}\ \textsc{Abs} \qquad \frac{C, \Gamma \vdash e_1 : \tau_2 \to \tau \qquad C, \Gamma \vdash e_2 : \tau_2}{C, \Gamma \vdash e_1\, e_2 : \tau}\ \textsc{App}$$

$$\frac{C, \Gamma \vdash \mathsf{ref} : \forall \alpha.\alpha \to \alpha\ \mathsf{ref}}{}\ \textsc{Ref} \qquad C, \Gamma \vdash := : \forall \alpha.\alpha\ \mathsf{ref} \to \alpha \to \alpha\ \ \textsc{Assign} \qquad C, \Gamma \vdash\ !\ : \forall \alpha.\alpha\ \mathsf{ref} \to \alpha\ \ \textsc{Deref}$$

$$\frac{C, \Gamma \vdash v : \sigma \qquad C, (\Gamma; x : \sigma) \vdash e : \tau}{C, \Gamma \vdash \mathsf{let}\, x = v\, \mathsf{in}\, e : \tau}\ \textsc{Let} \qquad \frac{C, \Gamma \vdash e : \tau \qquad C \Vdash \tau \leq \tau'}{C, \Gamma \vdash e : \tau'}\ \textsc{Sub}$$

$$\frac{C \wedge D, \Gamma \vdash v : \tau \qquad \bar{\alpha} \cap \mathsf{fv}(C, \Gamma) = \varnothing}{C \wedge \exists \bar{\alpha}.D, \Gamma \vdash v : \forall \bar{\alpha}[D].\tau}\ \forall\ \textsc{Intro} \qquad \frac{C, \Gamma \vdash v : \forall \bar{\alpha}[D].\tau \qquad C \Vdash [\bar{\tau}/\bar{\alpha}]D}{C, \Gamma \vdash v : [\bar{\tau}/\bar{\alpha}]\tau}\ \forall\ \textsc{Elim}$$

Figure 2.3: The system HM(X)

It is straightforward to check that, if $C, \Gamma \vdash e : \sigma$ is derivable, then $C \Vdash \sigma$ holds. This explains why the well-typedness of e can be determined by checking whether C alone is satisfiable; there is no need to inspect σ in addition.

For the type system to be safe, the semantics of constants, given by δ, must be correctly approximated by their types, given by Δ.

Definition 2.6 (δ-Typability) *Let C be satisfiable. We require that, for every constant* $\mathbf{c}$ *and closed value v, if $C, \Gamma \vdash \mathbf{c} : \tau_1 \to \tau_2$ and $C, \Gamma \vdash v : \tau_1$ hold, then $\delta(\mathbf{c}, v)$ is defined and $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2$ holds. We also require $C, \Gamma \vdash \mathbf{c} : \tau\ \mathsf{ref}$ to* not *hold.*

The following definition sums up the requirements that bear on every instance of the parameterized type system HM(X).

Definition 2.7 *An instance of HM(X) is defined by*

- *an extension of the type and constraint language, together with a standard interpretation, as specified in Definitions 2.1 and 2.2;*
- *a particular choice of the set of constants $Const$, together with functions δ and Δ, meeting the δ-typability requirement of Definition 2.6.*

As will be proven in section 2.3, any such instance of HM(X) enjoys syntactic type safety.

2.2 Preliminary results

2.2.1 Type substitutions

Sulzmann [40] gives two equivalent versions of the HM(X) type rules. In the one shown here, rule $\forall$ ELIM allows the universally quantified type variables to be instantiated using an arbitrary substitution. In the other version, not shown in this paper, rule $\forall$ ELIM requires these variables to be instantiated with the identity substitution, but a new rule appears ($\exists$-INTRO) which allows arbitrary substitutions to be encoded within a constraint. The two presentations are equivalent, that is, they give rise to the same valid judgements. As a result, it is enough to prove one of them correct.

Here, we adopt the substitution-based version. Accordingly, we must now demonstrate a series of results related to substitutions.

Definition 2.8 *A* substitution φ *is a finite mapping from type variables to types. A* renaming ϱ *is a bijective mapping from a finite set of type variables to itself. Substitutions and renamings are extended to total mappings from types to types, from constraints to constraints, and from type schemes to type schemes, in the natural, capture-avoiding manner.*

Lemma 2.1 *If $C \Vdash D$ then $\varphi(C) \Vdash \varphi(D)$. If $C \Vdash \sigma$, then $\varphi(C) \Vdash \varphi(\sigma)$.*

Lemma 2.2 *If φ_1 is idempotent and* $\mathrm{dom}(\varphi_2)$ *and* $\mathrm{fv}(\mathrm{rng}(\varphi_1)) \cup \mathrm{dom}(\varphi_1)$ *are disjoint then $\varphi_1 \circ \varphi_2 \circ \varphi_1 = \varphi_1 \circ \varphi_2$.*

Lemma 2.3 (Type Instantiation) *If there exists a derivation of $C, \Gamma \vdash e : \sigma$, then there exists a derivation of $\varphi(C), \varphi(\Gamma) \vdash e : \varphi(\sigma)$ with the same structure.*

Proof. By induction on the input derivation. We give only the key cases and follow the notations of figure 2.3. Note that the structure of the derivation is preserved by construction in the proof.

Cases VAR, SUB. By induction hypothesis and by Lemma 2.1.

Case $\forall$ INTRO. Without loss of generality, we may require $\bar{\alpha} \cap \text{fv}(\text{rng}(\varphi)) = \bar{\alpha} \cap \text{dom}(\varphi) = \varnothing$. Indeed, if such were not the case, one could apply the induction hypothesis to the premise and to a renaming which maps $\bar{\alpha}$ to fresh variables and does not affect any other variable free in the premise. Because the variables $\bar{\alpha}$ do not appear free in the conclusion, the latter would remain unchanged.

Now, let us apply the induction hypothesis to the premise and φ. This yields $\varphi(C) \wedge \varphi(D), \varphi(\Gamma) \vdash e : \varphi(\tau)$. From $\bar{\alpha} \cap \text{fv}(C, \Gamma) = \varnothing$ and the above requirement, we deduce $\bar{\alpha} \cap \text{fv}(\varphi(C), \varphi(\Gamma)) = \varnothing$. Thus, we may apply $\forall$ INTRO, which yields $\varphi(C) \wedge \exists\bar{\alpha}.\varphi(D), \varphi(\Gamma) \vdash e : \forall\bar{\alpha}[\varphi(D)].\varphi(\tau)$. Again, thanks to the above requirement, this is $\varphi(C \wedge \exists\bar{\alpha}.D), \varphi(\Gamma) \vdash e : \varphi(\forall\bar{\alpha}[D].\tau)$.

Case $\forall$ ELIM. Every substitution is the composition of an idempotent substitution and a renaming. Thus, we consider two sub-cases.

First, let us assume that φ is idempotent. By the induction hypothesis, we have that $\varphi(C), \varphi(\Gamma) \vdash e : \varphi(\forall\bar{\alpha}[D].\tau)$ holds. Without loss of generality we may assume it is the case that $\bar{\alpha} \cap \text{fv}(\text{rng}(\varphi)) = \varnothing$ and $\bar{\alpha} \cap \text{dom}(\varphi) = \varnothing$. (This follows from the fact that we identify type schemes modulo α-equivalence.) This yields $\varphi(C), \varphi(\Gamma) \vdash e : \forall\bar{\alpha}[\varphi(D)].\varphi(\tau)$ and (by Lemma 2.2) $\varphi \circ [\bar{\tau}/\bar{\alpha}] \circ \varphi = \varphi \circ [\bar{\tau}/\bar{\alpha}]$. Now, lemma 2.1 yields $\varphi(C) \Vdash \varphi([\bar{\tau}/\bar{\alpha}]D)$, that is, $\varphi(C) \Vdash \varphi \circ [\bar{\tau}/\bar{\alpha}](\varphi(D))$. Therefore, by $\forall$ ELIM, we obtain $\varphi(C), \varphi(\Gamma) \vdash e : \varphi \circ [\bar{\tau}/\bar{\alpha}](\varphi(\tau))$, that is, $\varphi(C), \varphi(\Gamma) \vdash e : \varphi([\bar{\tau}/\bar{\alpha}]\tau)$.

Second, let us assume that φ is a renaming ϱ. By applying the induction hypothesis to the premise, we obtain $\varrho C, \varrho\Gamma \vdash e : \varrho(\forall\bar{\alpha}[D].\tau)$, which can be written $\varrho C, \varrho\Gamma \vdash e : \forall(\varrho\bar{\alpha})[\varrho D].\varrho\tau$. Furthermore, Lemma 2.1 yields $\varrho C \Vdash \varrho[\bar{\tau}/\bar{\alpha}]D$, that is, $\varrho C \Vdash [\varrho\bar{\tau}/\varrho\bar{\alpha}]\varrho D$. Then, $\forall$ ELIM, applied to the substitution $[\varrho\bar{\tau}/\varrho\bar{\alpha}]$, yields $\varrho C, \varrho\Gamma \vdash e : [\varrho\bar{\tau}/\varrho\bar{\alpha}]\varrho\tau$, that is, $\varrho C, \varrho\Gamma \vdash e : \varrho[\bar{\tau}/\bar{\alpha}]\tau$. □

2.2.2 Normalization

In this section we define a normalized form for HM(X) type derivations. This normalization provides for a much easier analysis of type derivations in the subject reduction proof.

Lemma 2.4 *If* $\text{dom}(\varphi) \subseteq \bar{\alpha}$ *then* $\varphi(C) \Vdash \exists\bar{\alpha}.C$.

Lemma 2.5 *Any two consecutive instances of* $\forall$ INTRO *and* $\forall$ ELIM *may be suppressed.*

Proof. Suppose the following sequence appears in a derivation:

$$\dfrac{\dfrac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha} \cap \mathrm{fv}(C, \Gamma) = \varnothing}{C \wedge \exists\bar{\alpha}.D, \Gamma \vdash e : \forall\bar{\alpha}[D].\tau}\ (\forall\ \textsc{Intro}) \qquad C \wedge \exists\bar{\alpha}.D \Vdash [\bar{\tau}/\bar{\alpha}]D}{C \wedge \exists\bar{\alpha}.D, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau}\ (\forall\ \textsc{Elim})$$

From $C \wedge \exists\bar{\alpha}.D \Vdash [\bar{\tau}/\bar{\alpha}]D$, we may deduce $C \wedge \exists\bar{\alpha}.D \Vdash C \wedge [\bar{\tau}/\bar{\alpha}]D$. However, by Lemma 2.4, we have $[\bar{\tau}/\bar{\alpha}]D \Vdash \exists\bar{\alpha}.D$, so $C \wedge \exists\bar{\alpha}.D$ and $C \wedge [\bar{\tau}/\bar{\alpha}]D$ are equivalent. Furthermore, considering $\bar{\alpha} \cap \mathrm{fv}(C) = \varnothing$, we have $C \wedge [\bar{\tau}/\bar{\alpha}]D = [\bar{\tau}/\bar{\alpha}](C \wedge D)$. Similarly, $\bar{\alpha} \cap \mathrm{fv}(\Gamma) = \varnothing$ implies $[\bar{\tau}/\bar{\alpha}]\Gamma = \Gamma$. Now, Lemma 2.3, applied to the upper left judgement, yields $[\bar{\tau}/\bar{\alpha}](C \wedge D), [\bar{\tau}/\bar{\alpha}]\Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau$, which, according to the above arguments, is $C \wedge \exists\bar{\alpha}.D, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau$. The derivation of this judgement has the same structure as that of the upper left judgement, so these instances of $\forall$ INTRO and $\forall$ ELIM have effectively been suppressed. □

Lemma 2.6 (Normalization) *If $C, \Gamma \vdash e : \tau$ holds, then it must follow by* SUB *from a judgement $\mathcal{J}$ such that*

1. *if e is* $\mathsf{let}\, x = v \,\mathsf{in}\, e'$ *then $\mathcal{J}$ follows by* LET*;*
2. *if e is* $\mathsf{fix}\, z.\lambda x.e'$ *then $\mathcal{J}$ follows by* ABS*;*
3. *if e is $e_1\, e_2$ then $\mathcal{J}$ follows by* APP*;*
4. *if e is l then $\mathcal{J}$ follows by* LOC*;*
5. *if e is x then $\mathcal{J}$ follows by* VAR *and* $\forall$ ELIM*;*
6. *if e is* **c** *then $\mathcal{J}$ follows by* CONST *and* $\forall$ ELIM*;*
7. *if e is* ref *then $\mathcal{J}$ follows by* REF *and* $\forall$ ELIM*;*
8. *if e is* ! *then $\mathcal{J}$ follows by* DEREF *and* $\forall$ ELIM*;*
9. *if e is* := *then $\mathcal{J}$ follows by* ASSIGN *and* $\forall$ ELIM.

Proof. The judgement $C, \Gamma \vdash e : \tau$ must be the consequence of a syntax-directed rule, possibly followed by a sequence of instances of SUB, $\forall$ ELIM and $\forall$ INTRO.

By construction, $\forall$ INTRO cannot be followed by itself or by SUB. Lemma 2.5 shows that $\forall$ INTRO need never be followed by $\forall$ ELIM. Lastly, given the form of the judgement at hand,

$\forall$ INTRO cannot be the last rule in the derivation. It follows that $\forall$ INTRO need not appear at all in the sequence.

By construction, $\forall$ ELIM cannot follow itself or SUB, so the sequence must consist of at most one instance of $\forall$ ELIM, followed by a number of instances of SUB. By reflexivity and transitivity of entailment, the latter may be expanded or reduced to a single instance of SUB.

To conclude, notice that $\forall$ ELIM cannot follow LOC, LET, ABS or APP. $\square$

2.2.3 Value Substitution

In this section, we establish a classic *substitution* Lemma, which will be at the heart of the β- and let-reduction cases in the subject reduction proof. We begin with a *weakening* Lemma, which shows that a valid judgement remains valid under a stronger constraint.

Lemma 2.7 (Weakening) *$C, \Gamma \vdash e : \sigma$ and $C' \Vdash C$ imply $C', \Gamma \vdash e : \sigma$.*

Proof. By induction on the input derivation. We give only the key cases and follow the notations of figure 2.3.

Cases VAR, SUB and $\forall$ ELIM follow by transitivity of entailment.

Case $\forall$ INTRO. We have a deduction of the form

$$\frac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha} \cap \mathrm{fv}(C, \Gamma) = \varnothing}{C \wedge \exists\bar{\alpha}.D, \Gamma \vdash e : \forall\bar{\alpha}[D].\tau}$$

Without loss of generality, we may assume $\bar{\alpha} \cap \mathrm{fv}(C') = \varnothing$; if this were not the case, we could apply Lemma 2.3 to the first premise to make it so. Now, clearly $C' \wedge C \wedge D \Vdash C \wedge D$, so the induction hypothesis yields $C' \wedge C \wedge D, \Gamma \vdash e : \tau$. Furthermore, we have $\bar{\alpha} \cap \mathrm{fv}(C' \wedge C, \Gamma) = \varnothing$, therefore $\forall$ INTRO yields $C' \wedge C \wedge \exists\bar{\alpha}.D, \Gamma \vdash e : \forall\bar{\alpha}[D].\tau$. Lastly, by assumption, we have $C' \Vdash C \wedge \exists\bar{\alpha}.D$, so $C' = C' \wedge C \wedge \exists\bar{\alpha}.D$, therefore $C', \Gamma \vdash e : \forall\bar{\alpha}[D].\tau$ holds. $\square$

Lemma 2.8 (Substitution) *If $C, \Gamma; x : \sigma' \vdash e : \sigma$ and $C, \Gamma \vdash v : \sigma'$ then $C, \Gamma \vdash e[v/x] : \sigma$.*

Proof. By induction on the derivation of $C, \Gamma; x : \sigma' \vdash e : \sigma$. We give only the key cases.

Case $\forall$ INTRO. In this case $\sigma = \forall\bar{\alpha}[D].\tau$, $C = C' \wedge \exists\bar{\alpha}.D$ and we have a deduction of the form:

$$\frac{C' \wedge D, \Gamma; x : \sigma' \vdash e : \tau \qquad \bar{\alpha} \cap \mathrm{fv}(C', \Gamma; x : \sigma') = \varnothing}{C' \wedge \exists\bar{\alpha}.D, \Gamma; x : \sigma' \vdash e : \forall\bar{\alpha}[D].\tau}$$

By assumption we have that $C' \wedge \exists\bar{\alpha}.D, \Gamma \vdash v : \sigma'$ holds, and clearly $C' \wedge D \Vdash C' \wedge \exists\bar{\alpha}.D$, therefore by Lemma 2.7 we have $C' \wedge D, \Gamma \vdash v : \sigma'$. Then, by the induction hypothesis, $C' \wedge D, \Gamma \vdash e[v/x] : \tau$ holds. The result follows by $\forall$ INTRO.

Case VAR. Suppose that $e = x' \neq x$. Then $e[v/x] = e$ and $\Gamma(x') = (\Gamma; x : \sigma')(x')$, so the Lemma holds by VAR. Suppose on the other hand that $e = x$; then $e[v/x] = v$, so the lemma holds by assumption.

Case LET. In this case $e = \mathsf{let}\, x' = v' \,\mathsf{in}\, e'$, $\sigma = \tau$ and we have a deduction of the following form:

$$\frac{C, \Gamma; x : \sigma' \vdash v' : \sigma'' \qquad C, \Gamma; x : \sigma'; x' : \sigma'' \vdash e' : \tau}{C, \Gamma; x : \sigma' \vdash \mathsf{let}\, x' = v' \,\mathsf{in}\, e' : \tau}$$

By the induction hypothesis we have $C, \Gamma \vdash v'[v/x] : \sigma''$; and supposing that $x \neq x'$ it is the case that $\Gamma; x : \sigma'; x' : \sigma'' = \Gamma; x' : \sigma''; x : \sigma'$, hence we have also $C, \Gamma; x' : \sigma'' \vdash e'[v/x] : \tau$ by the induction hypothesis, so that $C, \Gamma \vdash \mathsf{let}\, x' = v'[v/x] \,\mathsf{in}\, e'[v/x] : \tau$ by LET, hence $C, \Gamma \vdash (\mathsf{let}\, x' = v' \,\mathsf{in}\, e')[v/x] : \tau$ by definition. On the other hand, if $x = x'$ then $\Gamma; x : \sigma'; x' : \sigma'' = \Gamma; x' : \sigma''$, so that $C, \Gamma; x' : \sigma'' \vdash e' : \tau$ by assumption, and since $C, \Gamma \vdash v'[v/x] : \sigma''$ by the preceding, the judgement $C, \Gamma \vdash \mathsf{let}\, x' = v'[v/x] \,\mathsf{in}\, e' : \tau$ holds by LET, therefore $C, \Gamma \vdash (\mathsf{let}\, x' = v' \,\mathsf{in}\, e')[v/x] : \tau$ by definition.

Case ABS. In this case $e = \mathsf{fix}\, z.\lambda x'.e'$, $\sigma = \tau_1 \to \tau_2$ and we have a deduction of the following form:

$$\frac{C, \Gamma; x : \sigma'; x' : \tau_1; z : \tau_1 \to \tau_2 \vdash e' : \tau_2}{C, \Gamma; x : \sigma' \vdash \mathsf{fix}\, z.\lambda x'.e' : \tau_1 \to \tau_2}$$

Supposing that $x \neq x'$ and $x \neq z$ it is the case that

$$\Gamma; x : \sigma'; x' : \tau_1; z : \tau_1 \to \tau_2 = \Gamma; x' : \tau_1; z : \tau_1 \to \tau_2; x : \sigma'$$

hence we have $C, \Gamma; x' : \tau_1; z : \tau_1 \to \tau_2 \vdash e'[v/x] : \tau_2$ by the induction hypothesis, so $C, \Gamma \vdash \mathsf{fix}\, z.\lambda x'.(e'[v/x]) : \tau_1 \to \tau_2$ by ABS, therefore $C, \Gamma \vdash (\mathsf{fix}\, z.\lambda x'.e')[v/x] : \tau_1 \to \tau_2$ by definition. On the other hand, supposing that $x = x'$ it is the case that

$$\Gamma; x : \sigma'; x' : \tau_1; z : \tau_1 \to \tau_2 = \Gamma; x' : \tau_1; z : \tau_1 \to \tau_2$$

and since $C, (\Gamma; x : \sigma'; x' : \tau_1; z : \tau_1 \to \tau_2) \vdash e' : \tau_2$ by assumption therefore $C, (\Gamma; x' : \tau_1; z : \tau_1 \to \tau_2) \vdash e' : \tau_2$, so $C, \Gamma \vdash \mathsf{fix}\, z.\lambda x'.e' : \tau_1 \to \tau_2$ by ABS, thus $C, \Gamma \vdash (\mathsf{fix}\, z.\lambda x'.e')[v/x] : \tau_1 \to \tau_2$ by definition. The case in which $x = z$ follows similarly. $\square$

Lemma 2.9 (Substitution for functions) *Let* $\Gamma' = (\Gamma; x : \tau'; z : \tau' \to \tau)$. *If* $C, \Gamma' \vdash e : \tau$ *and* $C, \Gamma \vdash v : \tau'$, *then* $C, \Gamma \vdash e[v/x][\mathsf{fix}\, z.\lambda x.e/z] : \tau$.

Proof. By ABS and two consecutive applications of Lemma 2.8.

2.3 Central Results

In this section we demonstrate the type soundness results for HM(X), specifically subject reduction, progress and type safety.

Definition 2.9 *In order to properly state subject reduction, type judgements are extended to configurations:*

$$\textsc{Config}\quad \frac{C, \Gamma \vdash e : \tau \qquad \forall l \in \mathrm{dom}(\Gamma) \quad C, \Gamma \vdash \sigma(l) : \Gamma(l)}{C, \Gamma \vdash e, \sigma : \tau}$$

A configuration e, σ *is* well-typed *if there exists a judgement* $C, \Gamma \vdash e, \sigma : \tau$ *deducible by* CONFIG, *with* C *satisfiable; such a judgement is* valid.

Theorem 2.1 (Subject Reduction) *Let* C *be satisfiable. If* $C, \Gamma \vdash e_1, \sigma_1 : \tau$ *is derivable and* $e_1, \sigma_1 \to e_2, \sigma_2$, *then, for some* Γ' *which extends* Γ *with bindings for new memory locations,* $C, \Gamma' \vdash e_2, \sigma_2 : \tau$ *is derivable.*

Proof. By induction on the definition of the reduction relation (see figure 2.2).

According to Lemma 2.6, the derivation of $C, \Gamma \vdash e_1 : \tau$ ends with an instance of SUB, which we will disregard, without loss of generality. (Indeed, we then have $C, \Gamma \vdash e_1 : \tau'$ and $C \Vdash \tau' \leq \tau$; once we have proven $C, \Gamma \vdash e_2 : \tau'$, applying SUB again shall yield $C, \Gamma \vdash e_2 : \tau$, as desired.)

For reduction cases which do not affect the store, it is sufficient to prove that $C, \Gamma \vdash e_2 : \tau$ is derivable to demonstrate the result.

Case (δ). Then, e_1 is $\mathbf{c}\, v$ and e_2 is $\delta(\mathbf{c}, v)$. By Lemma 2.6 we have a sub-derivation of the following form:

$$\frac{C, \Gamma \vdash \mathbf{c} : \tau_1 \to \tau \qquad C, \Gamma \vdash v : \tau_1}{C, \Gamma \vdash \mathbf{c}\, v : \tau}$$

Then, according to Definition 2.6, $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau$ holds.

Case (β). Then, e_1 is $(\mathsf{fix}\, z.\lambda x.e)\, v$ and e_2 is $e[v/x][\mathsf{fix}\, z.\lambda x.e/z]$. By Lemma 2.6 we have a sub-derivation of the following form:

$$\dfrac{\dfrac{\dfrac{C, \Gamma; x : \tau_1'; z : \tau_1' \to \tau' \vdash e : \tau'}{C, \Gamma \vdash \mathsf{fix}\, z.\lambda x.e : \tau_1' \to \tau'} \qquad C \Vdash \tau_1' \to \tau' \leq \tau_1 \to \tau}{C, \Gamma \vdash \mathsf{fix}\, z.\lambda x.e : \tau_1 \to \tau} \qquad C, \Gamma \vdash v : \tau_1}{C, \Gamma \vdash (\mathsf{fix}\, z.\lambda x.e)\, v : \tau}$$

Now, $C \Vdash \tau_1' \to \tau' \leq \tau_1 \to \tau$ implies $C \Vdash \tau_1 \leq \tau_1'$ and $C \Vdash \tau' \leq \tau$. Therefore $C, \Gamma \vdash v : \tau_1'$ by assumption and SUB; and since $C, (\Gamma; x : \tau_1'; z : \tau_1' \to \tau') \vdash e : \tau'$ by assumption, therefore $C, \Gamma \vdash e[v/x][\mathsf{fix}\, z.\lambda x.e/z] : \tau'$ by Lemma 2.9. By SUB, $C, \Gamma \vdash e[v/x][\mathsf{fix}\, z.\lambda x.e/z] : \tau$ follows.

Case (*let*). Then, e_1 is $\mathsf{let}\, x = v\, \mathsf{in}\, e$ and e_2 is $e[v/x]$. By Lemma 2.6 we have a sub-derivation of the following form:

$$\dfrac{C, \Gamma; x : \sigma \vdash e : \tau \qquad C, \Gamma \vdash v : \sigma}{C, \Gamma \vdash \mathsf{let}\, x = v\, \mathsf{in}\, e : \tau}$$

By Lemma 2.8, we obtain $C, \Gamma \vdash e[v/x] : \tau$.

Case (*deref*). Then, e_1 is $!\, l$ and e_2 is $\sigma_1(l)$. By Lemma 2.6, we have a sub-derivation of the following form:

$$\dfrac{\dfrac{\begin{array}{c} C, \Gamma \vdash\, ! : \tau' \,\text{ref} \to \tau' \\ C \Vdash \tau' \,\text{ref} \to \tau' \leq \tau_1 \,\text{ref} \to \tau \end{array}}{C, \Gamma \vdash\, ! : \tau_1 \,\text{ref} \to \tau} \qquad \dfrac{\dfrac{\Gamma(l) = \tau''}{C, \Gamma \vdash l : \tau'' \,\text{ref}} \qquad C \Vdash \tau'' \,\text{ref} \leq \tau_1 \,\text{ref}}{C, \Gamma \vdash l : \tau_1 \,\text{ref}}}{C, \Gamma \vdash\, !\, l : \tau}$$

By CONFIG, $C, \Gamma \vdash \sigma(l) : \tau''$ is derivable. and by properties of $\leq$ we have $C \Vdash \tau_1 \leq \tau$ and $C \Vdash \tau'' \leq \tau_1$. Thus, by transitivity of $\leq$ we have $C \Vdash \tau'' \leq \tau$, so $C, \Gamma \vdash \sigma(l) : \tau$ can be derived by SUB.

Case (*ref*). The reduction is $\text{ref}\, v, \sigma_1 \to l, \sigma_1[l \mapsto v]$, where $l \notin \text{dom}(\sigma_1)$. By Lemma 2.6 we have a sub-derivation of the following form:

$$\dfrac{\dfrac{C, \Gamma \vdash \text{ref} : \tau' \to \tau' \,\text{ref} \qquad C \Vdash \tau' \to \tau' \,\text{ref} \leq \tau_2 \to \tau}{C, \Gamma \vdash \text{ref} : \tau_2 \to \tau} \qquad C, \Gamma \vdash v : \tau_2}{C, \Gamma \vdash \text{ref}\, v : \tau}$$

These imply $C \Vdash \tau_2 \le \tau'$ and $C \Vdash \tau' \,\mathrm{ref} \le \tau$. Define Γ' as $(\Gamma; l : \tau')$. By LOC and SUB, $C, \Gamma' \vdash l : \tau$ holds. Furthermore, since $C, \Gamma \vdash v : \tau_2$ holds and since v is $\sigma_2(l)$, SUB yields $C, \Gamma \vdash \sigma_2(l) : \tau'$. Because l is fresh, this implies $C, \Gamma' \vdash \sigma_2(l) : \tau'$. Lastly, l's freshness and CONFIG yield $C, \Gamma' \vdash l, \sigma_2 : \tau$.

Case (*assign*). The reduction is $:= l\, v, \sigma_1 \to v, \sigma_1[l \mapsto v]$, where $l \in \mathrm{dom}(\sigma_1)$. By Lemma 2.6, we have a sub-derivation of the following form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c} C,\Gamma \vdash := \,:\, \tau'\,\mathrm{ref} \to \tau' \to \tau' \\ C \Vdash \tau'\,\mathrm{ref} \to \tau' \to \tau' \le \tau_1 \to \tau_2 \to \tau_3 \end{array}}{C,\Gamma \vdash := \,:\, \tau_1 \to \tau_2 \to \tau_3} \qquad \dfrac{\begin{array}{c} \dfrac{\Gamma(l) = \tau''}{C,\Gamma \vdash l : \tau''\,\mathrm{ref}} \\ C \Vdash \tau''\,\mathrm{ref} \le \tau_1 \end{array}}{C,\Gamma \vdash l : \tau_1}}{\begin{array}{c} C,\Gamma \vdash := l : \tau_2 \to \tau_3 \\ C \Vdash \tau_2 \to \tau_3 \le \tau_2' \to \tau \end{array}}}{C,\Gamma \vdash := l : \tau_2' \to \tau} \qquad C,\Gamma \vdash v : \tau_2'}{C,\Gamma \vdash := l\, v : \tau}$$

From these, we deduce $C \Vdash \tau_2' \le \tau_2$ and $C \Vdash \tau_2 \le \tau'$. Furthermore, we find $C \Vdash \tau''\,\mathrm{ref} \le \tau_1 \le \tau'\,\mathrm{ref}$, which implies $C \Vdash \tau' \le \tau''$. As a result, by SUB, $C, \Gamma \vdash v : \tau''$ holds, i.e. $C, \Gamma \vdash \sigma_2(l) : \tau''$ in this case is derivable. Furthermore, we find $C \Vdash \tau' \le \tau_3$ and $C \Vdash \tau_3 \le \tau$, hence $C, \Gamma \vdash v : \tau$ is derivable by SUB. The result follows by CONFIG.

Case $E[e_1], \sigma_1 \to E[e_2], \sigma_2$, where $e_1, \sigma_1 \to e_2, \sigma_2$. This case follows by the induction hypothesis and a simple "replacement" Lemma, analogous to that found in [47], except newly created memory locations must be taken into account. ☐

To demonstrate progress, rather than defining a class of *faulty* expressions that approximates the class of stuck expressions, and proving a *uniform evaluation* result as in e.g. [47], we adopt the more direct method of [25] and demonstrate the following:

Lemma 2.10 (Progress) *If a closed configuration e, σ is well-typed and irreducible, then e is a value.*

Proof. Suppose on the contrary that e, σ is well-typed and irreducible, but e is not a value. Then e is of the form $E[f]$, with f also well-typed as a precedent of a valid instance of CONFIG, where one of the following cases holds:

1. f is of the form $\mathbf{c}\, v$ and $\delta(\mathbf{c}, v)$ is undefined. Now, if $\mathbf{c}\, v$ is well-typed, then by Lemma 2.6 there exists a judgement that follows by APP with valid precedents $C, \Gamma \vdash \mathbf{c} : \tau_1 \to \tau_2$ and

$C, \Gamma \vdash v : \tau_1$. But then by Definition 2.6 it must be the case that $\delta(\mathbf{c}, v)$ is defined, which is a contradiction.

2. f is of the form $l\, v$. By Lemma 2.6 there exists a judgement that follows by APP with valid precedent $C, \Gamma \vdash l : \tau_1 \to \tau_2$. By Lemma 2.6, this judgement must follow from LOC and SUB, so we have $C \Vdash \tau'$ ref $\leq \tau_1 \to \tau_2$, which is a contradiction.

3. f is of the form $:= v$ or $!\, v$ where v is not a memory location. In either case, by applications of Lemma 2.6, we have $C, \Gamma \vdash v : \tau$ ref. According to Definition 2.6, v cannot be a constant. One checks that all other value forms must have functional type, that is, we must have $C \Vdash \tau_1 \to \tau_2 \leq \tau$ ref, again a contradiction.

4. f is of the form $:= l\, v$ or $:= l$ and $l \notin \mathrm{dom}(\sigma)$. f is well-typed, so $l \in \mathrm{dom}(\Gamma)$; then, CONFIG requires $\sigma(l)$ to be defined, a contradiction. □

We may now state and prove progress and type safety. In order to do so, we make the usual Definitions:

Definition 2.10 *If $e, \varnothing \to^\star e', \sigma'$, where e', σ' is irreducible but e' is not a value, then e is said to* go wrong.

Theorem 2.2 (Type Safety) *If e is closed and well-typed, then e does not go wrong.*

Proof. Suppose that $e, \varnothing$ reduces to e', σ' and the latter is irreducible. Since e is well-typed, there exists a derivable judgement $C, \Gamma \vdash e, \varnothing : \tau$ with C satisfiable. Then, by repeated application of Theorem 2.1, we have $C, \Gamma' \vdash e', \sigma' : \tau$, for some Γ'. Then, by Lemma 2.10, e' is a value. □

2.4 Type inference

In this section we define type inference for HM(X). As we will see, type inference for the framework is defined modulo a constraint *normalization* procedure, just as HM(X) type judgements are defined modulo the specification of a constraint system. A constraint normalization procedure is essentially a constraint solution algorithm that must be specified for each particular instance of HM(X), and is really the heart of type inference. For example, *unification* is a normalization procedure for an equality constraint system. Since the general HM(X) type inference algorithm is proven correct in [39], proving correctness of inference for an instance of the framework requires

$$\frac{\Delta_\sigma(\mathbf{c}_\sigma) = \forall\bar{\alpha}[D].\tau \qquad \bar{\beta}\text{ fresh} \qquad (C,\varphi) = \mathit{normalize}(C,[\bar{\beta}/\bar{\alpha}])}{\varphi|_{\mathrm{fv}(\Gamma)}, C, \Gamma \vdash^W \mathbf{c}_\sigma : \varphi(\tau)}\ \textsc{Const}_\sigma$$

$$\frac{\Gamma(x) = \forall\bar{\alpha}[D].\tau \qquad \bar{\beta}\text{ fresh} \qquad (C,\varphi) = \mathit{normalize}(C,[\bar{\beta}/\bar{\alpha}])}{\varphi|_{\mathrm{fv}(\Gamma)}, C, \Gamma \vdash^W x : \varphi(\tau)}\ \textsc{Var}$$

$$\frac{\begin{array}{c}\varphi', C', (\Gamma; x:\alpha; z:\beta) \vdash^W e : \tau \qquad \alpha,\beta\text{ fresh}\\ D = C' \wedge (\alpha \to \tau \leq \beta) \qquad (C,\varphi) = \mathit{normalize}(D,\varphi')\end{array}}{\varphi\backslash\{\alpha,\beta\}, C, \Gamma \vdash^W \mathsf{fix}\, z.\lambda x.e : \varphi(\beta)}\ \textsc{Abs}$$

$$\frac{\begin{array}{c}\varphi_1, C_1, \Gamma \vdash^W e_1 : \tau_1 \qquad \varphi_2, C_2, \Gamma \vdash^W e_2 : \tau_2\\ \varphi' = \varphi_1 \sqcup \varphi_2 \qquad D = C_1 \wedge C_2 \wedge (\tau_1 \leq \tau_2 \to \alpha) \qquad \alpha\text{ fresh} \qquad (C,\varphi) = \mathit{normalize}(D,\varphi')\end{array}}{\varphi|_{\mathrm{fv}(\Gamma)}, C, \Gamma \vdash^W e_1 e_2 : \varphi(\alpha)}\ \textsc{App}$$

$$\frac{\begin{array}{c}\varphi_1, C_1, \Gamma \vdash^W e : \tau \qquad (C_2,\sigma) = \mathit{gen}(C_1, \varphi_1(\Gamma), \tau) \qquad \varphi_2, C_3, \Gamma; x:\sigma \vdash^W e' : \tau'\\ \varphi' = \varphi_1 \sqcup \varphi_2 \qquad D = C_2 \wedge C_3 \qquad (C,\varphi) = \mathit{normalize}(D,\varphi')\end{array}}{\varphi|_{\mathrm{fv}(\Gamma)}, C, \Gamma \vdash^W \mathsf{let}\, x = e\, \mathsf{in}\, e' : \varphi(\tau')}\ \textsc{Let}$$

Figure 2.4: HM(X) type inference

only a proof of correctness of the instance's normalization procedure. This is another benefit of the HM(X) framework.

Type inference for HM(X) is defined in Fig. 2.4; this definition is based almost entirely on the algorithm proposed in [23], with straightforward additions to handle recursive binding and state operations. In Fig. 2.4, state operations are treated as primitives for brevity: The symbol $\mathbf{c}_\sigma$ ranges over constants $\mathbf{c}$ in an instance of HM(X), along with the operators ref, := and !. The symbol

$$\textsc{Sub}^{\preccurlyeq}\ \frac{C \Vdash \tau \leq \tau'}{C \vdash^i \tau \preccurlyeq \tau'} \qquad (\preccurlyeq\forall)\ \frac{C \wedge D \vdash^i \sigma \preccurlyeq \tau \qquad \bar{\alpha} \cap \mathrm{fv}(C, \sigma) = \varnothing}{C \wedge \exists \alpha. D \vdash^i \sigma \preccurlyeq \forall \bar{\alpha}[D].\tau}$$

$$(\forall\preccurlyeq)\ \frac{C \vdash^i [\bar{\tau}/\bar{\alpha}]\tau \preccurlyeq \sigma' \qquad C \Vdash [\bar{\tau}/\bar{\alpha}]D}{C \vdash^i \forall \bar{\alpha}[D].\tau \preccurlyeq \sigma'}$$

Figure 2.5: HM(X) type scheme instance relation

Δ_σ denotes the initial binding environment Δ, augmented with the following bindings:

$$\begin{array}{rcl} \mathrm{ref} & : & \forall \alpha.\alpha \rightarrow \alpha\ \mathrm{ref} \\ := & : & \forall \alpha.\alpha\ \mathrm{ref} \rightarrow \alpha \rightarrow \alpha \\ ! & : & \forall \alpha.\alpha\ \mathrm{ref} \rightarrow \alpha \end{array}$$

To completely define type inference, we must also define the operator $\sqcup$, and specify the behavior of the functions $normalize$ and gen that occur in the definition, which we accomplish in the following. Note that these will be *specifications*, not definitions; if type inference is desired, an instantiation of HM(X) must include definitions of these operations that satisfy the specifications for the relevant constraint systems.

We begin by defining an ordering $\leq$ on substitutions; as Sulzmann observes in [40], it follows from results presented in [18] that this ordering induces a complete lower semi-lattice where least upper bounds, if they exist, correspond to unification.

Definition 2.11 *$\varphi_1 \leq \varphi_2$ iff* $\mathrm{dom}(\varphi_1) \subseteq \mathrm{dom}(\varphi_2)$ *and there exists φ such that $\varphi \circ \varphi_1 = \varphi_2$. We denote the least upper bound of φ_1 and φ_2 as $\varphi_1 \sqcup \varphi_2$.*

Now, we specify the expected behavior of the gen function, which must yield a "maximally generalized" type scheme with respect to a given constraint and type environment:

Definition 2.12 *The function gen but must satisfy the following equation:*

$$gen(C, \Gamma, \tau) = (D \wedge \exists \bar{\alpha}.C', \forall \bar{\alpha}[C'].\tau)$$

where $C = C' \wedge D$ and $\bar{\alpha} = (\mathrm{fv}(\tau) \cup \mathrm{fv}(C'))\backslash \mathrm{fv}(\Gamma)$ and $\bar{\alpha} \cap \mathrm{fv}(D) = \varnothing$.

To specify the $normalize$ function, we must first define the meaning of *normal forms* and *principal normal form* of types and constraints, since that is what $normalize$ will be expected to compute:

Definition 2.13 (C_1, φ_1) *is a* normal form *of* (C_2, φ_2) *iff* $\varphi_1 \leq \varphi_2$*,* $C_1 \Vdash \varphi_1(C_2)$ *and* $\varphi(C) = C$*.* (C_1, φ_1) *is a* principal normal form *of* (C_2, φ_2) *iff for all normal forms* (C, φ) *of* (C_2, φ_2) *it is the case that* $\varphi_2 \leq \varphi$ *and* $C \Vdash \varphi(C_1)$*.*

Now we may specify the expected behavior of the $normalize$ function, as follows:

Definition 2.14 *The function* $normalize$ *satisfies the following equations:*

$$
\begin{array}{rcll}
normalize(C_1, \varphi_1) & = & (C_2, \varphi_2) & \text{if } (C_2, \varphi_2) \text{ is a principal normal form of } (C_1, \varphi_1) \\
 & = & \textbf{fail} & \text{if no principal normal form of } (C_1, \varphi_1) \text{ exists}
\end{array}
$$

Given these definitions, we may now state the HM(X) type inference correctness results proved in [39], which assume an instantiation of the framework that includes definitions of gen and $normalize$ that satisfy specifications. While these results cover a version of the inference algorithm without recursive binding and state operations, we believe they can be easily extended to accommodate them. First, we state the soundness result, which says that an inferred type is a valid type:

Theorem 2.3 (Soundness of HM(X) Inference) *Given* e *and* Γ*, if* $\varphi, C, \Gamma \vdash^W e : \tau$ *then the judgement* $C, \varphi(\Gamma) \vdash e : \tau$ *is valid, with* $\varphi(C) = C$ *and* $\varphi(\tau) = \tau$*.*

To state the completeness result for inference, we must also define a type scheme instance relation, since the result states that if an expression is well-typed, then a most general type is inferred for it; this relation is defined in Fig. 2.5. Thus:

Theorem 2.4 (Completeness of HM(X) Inference) *If the judgement* $C, \varnothing \vdash e : \sigma$ *is valid, then* $\varphi, C', \varnothing \vdash^W e : \tau$ *where* $gen(C', \varnothing, \tau) = (C'', \sigma')$ *and there exists* φ' *such that* $C \Vdash \varphi'(C'')$ *and* $C \vdash^i \varphi'(\sigma') \preccurlyeq \sigma$*.*

2.4.1 OCaml implementation

An OCaml implementation of the HM(X) inference algorithm is included in the Appendix, in the module `Hmx`. The functor `Hmx.Make` is parameterized by modules `G`, `X` and `P`, where `G : Ground.Signature`, `X : ConstraintSystem.S` and `P : Primitives.S`; these signatures are also included in the appendix. Modules matching signature `Ground.Signature` implement the HM(X) core type language. Modules matching signature `ConstraintSystem.S` implement the constraint system for an instance of HM(X). Note that any implementation of `ConstraintSystem.S` must include a function `constrain`, which conjoins a new constraint to a pre-existing one, and solves the new constraint; this is the implementation of the *normalize* function. Modules matching signature `Primitives.S` implement the initial type bindings for any additional constants in an instance of HM(X), in the type and constraint language specified by `G` and `X`. This module is also expected to implement the bindings for state operations, which are not included in the module `Hmx` for simplicity; in particular, note that it is the responsibility of `P` to implement type schemes, so expecting `Hmx` to implement these bindings would be out of order.

Chapter 3

Technical Interlude: the Language pml_B

In this chapter we present the pml_B programming language, which includes primitive records, sets, and associated operations, and a static type discipline for the language that provides accurate specifications of these constructs. The language and type system is defined as an instantiation of the HM(X) framework described in the previous chapter. In the following chapters, we will use pml_B as a target language for transformations of our refigured stack inspection language, to be defined in Chapter 4 and a capability-based security language to be defined in Chapter 5. These transformations, and the pml_B type system, will serve as the foundation for the development of type systems in the source languages.

The pml_B language of records includes default values in the style of Rémy's Projective ML [30]. The language of sets includes syntax for defining sets of urelements—that is, atomic elements— as well as operations such as intersection, union, difference, etc. Sets are at first approximation records, where all values are of trivial type `unit`. However, since sets are simpler than records, there are set operations which can be effectively modeled statically that are difficult or impossible in the case of records, and set types can also be simpler than record types. We equip the language with a type system that accurately specifies the contents of records, sets, and the results of associated operations; we also show that this type system is sound. To define the type system, we instantiate HM(X) with a constraint system containing *row types* [32] and *conditional constraints* [24]. Row types were originally developed for application to extensible records with default values; we show here how they can also be used to type sets which include new operations not defined for record row types.

$x \in \mathcal{V},\ a \in \mathcal{L}_a,\ b \in \mathcal{L}_b,\ B \subseteq \mathcal{L}_b$			*identifiers*
v	$::=$	$\mathsf{fix}\, z.\lambda x.e \mid s \mid \{v\} \mid v\{a = v\} \mid \mathsf{ref} \mid {:=} \mid ({:=}l) \mid !$	*values*
s	$::=$	$B \mid \bar{B} \mid \vee \mid \wedge \mid \ominus \mid \ni_b \mid ?_b$	*sets, set operations*
e	$::=$	$x \mid v \mid e\, e \mid \mathsf{let}\, x = v\, \mathsf{in}\, e \mid \{e\} \mid e\{a = e\} \mid e.a$	*expressions*
E	$::=$	$[\,] \mid E\, e \mid v\, E \mid \{E\} \mid E\{a = e\} \mid v\{a = E\} \mid E.a$	*evaluation contexts*

Figure 3.1: Grammar for pml_B

3.1 The pml_B language definition

In this section, we formally define the pml_B language syntax and operational semantics. In Sect. 3.3 the semantics will be trivially re-figured as an instance of HM(X), with sets, records and operations defined as language constants with δ-rules conforming to the operational rules presented here.

3.1.1 Syntax

The grammar for pml_B is given in Fig. 3.1. The language is based on Rémy's Projective ML [30], containing records with default values, manipulated with the *elevation* and *modification* record constructors $\{e\}$ and $e\{a = e'\}$, and the *projection* destructor $e.a$.

The language allows definition of finite sets B of urelements $b \in \mathcal{L}_b$ where each b can be considered an arbitrary identifier. Countably infinite cosets $\bar{B}$ may also be defined. This latter feature presents some practical implementation issues, but in this presentation we take it at mathematical "face value"— that is, we take $\bar{B}$ to denote $\mathcal{L}_b \backslash B$. Basic set operations are provided, including $\ni_b$, $\wedge$, $\vee$ and $\ominus$, which are membership check, intersection, union and difference operations, respectively. For technical reasons, the difference operation removes elements in the first argument from elements in the second argument, which is perhaps an inversion of the expected behavior, but this will be convenient for our presentation. Also provided is a set membership test operation $?_b$, that allows branching on the presence or absence of a set element in a given set, as opposed to failure in the case of absence à la $\ni_b$. For clarity of presentation, we define the following

$$
\begin{array}{rcllr}
(\mathsf{fix}\, z.\lambda x.e)v, \sigma & \to & e[v/x][\mathsf{fix}\, z.\lambda x.e/z], \sigma & & (\beta) \\
\mathsf{let}\, x = v \,\mathsf{in}\, e, \sigma & \to & e[v/x], \sigma & & (\textit{let}) \\
\mathrm{ref}\, v, \sigma & \to & l, \sigma[l \mapsto v] & l \notin \mathrm{dom}(\sigma) & (\textit{ref}) \\
l := v, \sigma & \to & v, \sigma[l \mapsto v] & l \in \mathrm{dom}(\sigma) & (\textit{assign}) \\
!l, \sigma & \to & \sigma(l), \sigma & & (\textit{bang}) \\
\{v\}.a, \sigma & \to & v, \sigma & & (\textit{default}) \\
v_1\{a = v_2\}.a, \sigma & \to & v_2, \sigma & & (\textit{access}) \\
v_1\{a' = v_2\}.a, \sigma & \to & v_1.a, \sigma & a' \neq a & (\textit{skip}) \\
B \ni b, \sigma & \to & B, \sigma & \text{if } b \in B & (\textit{memcheck}) \\
B_1 \wedge B_2, \sigma & \to & B_1 \cap B_2, \sigma & & (\textit{intersect}) \\
B_1 \vee B_2, \sigma & \to & B_1 \cup B_2, \sigma & & (\textit{union}) \\
B_1 \ominus B_2, \sigma & \to & B_1 \backslash B_2, \sigma & & (\textit{difference}) \\
?_b\, B, \sigma & \to & \lambda f.\lambda g.f(B), \sigma & \text{if } b \in B & (\textit{memtesty}) \\
?_b\, B, \sigma & \to & \lambda f.\lambda g.g(B), \sigma & \text{if } b \notin B & (\textit{memtestn}) \\
E[e], \sigma & \to & E[e'], \sigma' & \text{if } e, \sigma \to e', \sigma' & (\textit{context})
\end{array}
$$

Figure 3.2: Operational semantics for pml_B

syntactic sugar:

$$
\begin{array}{rcl}
(\ni_b\, e) & \triangleq & (e \ni b) \\
(\wedge e_1 e_2) & \triangleq & (e_1 \wedge e_2) \\
(\vee e_1 e_2) & \triangleq & (e_1 \vee e_2) \\
(\ominus e_1 e_2) & \triangleq & (e_2 \ominus e_1)
\end{array}
$$

3.1.2 Operational semantics

The operational semantics for pml_B is given in Fig. 3.2. As is the case for the core HM(X) system, it is defined as a relation $\to$ on pairs e, σ, where stores σ and operations on stores are as defined in Chapter 2. Since pml_B will be defined as an instance of HM(X), this incorporation of state into the language is trivial.

The reflexive, transitive closure of $\to$ is denoted $\to^\star$. Stuck expressions and going wrong

$$
\begin{array}{rcll}
\tau & ::= & \alpha, \beta, \ldots \mid \tau \to \tau \mid \{\tau\} \mid \{\cdot\tau\cdot\} \mid \ell : \tau\,;\ \tau \mid \partial\tau \mid \tau \text{ ref} \mid c & \textit{types} \\
c & ::= & + \mid - \mid \top \mid \bot & \textit{constructors} \\
C & ::= & \mathbf{true} \mid C \wedge C \mid \exists\alpha.C \mid \tau = \tau \mid \tau \leq \tau \mid \text{if } c \leq \tau \text{ then } \tau \leq \tau & \textit{constraints}
\end{array}
$$

Figure 3.3: RS grammar

are as defined in Chapter 2. The operational rules for λ_{sec} are relatively straightforward, with each operation defined as one might expect.

3.2 The type constraint system RS

We define the type system for pml_B via instantiation of the HM(X) framework discussed in Chapter 2; in this section we define the constraint system which parameterizes that instantiation, called RS. The system RS comprises row types and conditional constraints. Following the guidelines specified in Definition 2.7, the definition includes the type and constraint language itself (Sect. 3.2.1), together with its logical interpretation in a model (Sect. 3.2.2 and Sect. 3.2.3).

3.2.1 The type and constraint language

The syntax of types and constraints is defined in Fig. 3.3. The syntax contains language for expressing *record* and *set* types (hence the name RS: <u>R</u>ecords and <u>S</u>ets).

To describe the contents of sets and records, we use *rows*. Row types are built up using the usual constructors, including $\partial\tau$ which specifies that all fields not otherwise mentioned in a row have type τ. In Fig. 3.3 and henceforth, we let ℓ range over elements of $\mathcal{L}_b \cup \mathcal{L}_a$. The original presentation of rows [30, 32] includes an equational theory, which in particular allow rows to commute. Here these equations are not axiomatic, but rather they hold as a result of the interpretation defined in Sect. 3.2.3.

Record types are built up from row types using the record type constructor $\{\cdot\}$. Set types are also built up from a particular form of row types, using the set type constructor $\{\cdot\cdot\cdot\}$. These particular row types are built up from *presence* constructors, which specify whether a given element may be present in a set ($+$), may not be present in it ($-$), may or may not appear in it ($\top$), or whether this information is irrelevant, because the set itself is unavailable ($\bot$) (*NB*: $\bot$ and $\top$ here are *not* the same as the "top" and "bottom" types in non-structural subtyping systems!). This form

$$\frac{\alpha \in \mathcal{V}_k}{\alpha : k} \qquad \frac{\tau : \mathit{Type}}{\tau\ \mathrm{ref} : \mathit{Type}} \qquad \frac{\tau, \tau' : \mathit{Type}}{\tau \to \tau' : \mathit{Type}} \qquad \frac{\tau : \mathit{Row}(\tau)_\varnothing}{\{\tau\} : \mathit{Type}} \qquad \frac{\tau : \mathit{Row}(c)_\varnothing}{\{\cdot\tau\cdot\} : \mathit{Type}}$$

$$\frac{\tau : \mathit{Type}}{\partial\tau : \mathit{Row}(\tau)_A} \qquad \frac{\tau : \mathit{Con}}{\partial\tau : \mathit{Row}(c)_B} \qquad c : \mathit{Con}$$

$$\frac{\tau : \mathit{Con} \qquad b \notin B \qquad \tau' : \mathit{Row}(c)_{B \cup \{b\}}}{(b : \tau\ ;\ \tau') : \mathit{Row}(c)_B} \qquad \frac{\tau : \mathit{Type} \qquad a \notin A \qquad \tau' : \mathit{Row}(\tau)_{A \cup \{a\}}}{(a : \tau\ ;\ \tau') : \mathit{Row}(\tau)_A}$$

Figure 3.4: Type kinding rules for RS types

$$\frac{}{\vdash \mathbf{true}} \qquad \frac{\vdash C_1, C_2}{\vdash C_1 \wedge C_2} \qquad \frac{\vdash C}{\vdash \exists\alpha.C} \qquad \frac{\tau, \tau' : k}{\begin{array}{c}\vdash \tau = \tau' \\ \vdash \tau \leq \tau'\end{array}}$$

$$\frac{\tau : \mathit{Con} \qquad \vdash C}{\vdash \text{if } c \leq \tau \text{ then } C} \qquad \frac{\tau, \tau', \tau'' : \mathit{Row}(c)_B}{\vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''}$$

Figure 3.5: Type kinding rules for RS constraints

is enforced by the kinding rules, defined below. We will also define a succinct, more readable form of set types in Sect. 3.2.4, which are defined as syntactic sugar for the primitive form. A significant consequence of this primitive definition of set types, as being built up from a specific kind of rows, is that set types can be soundly implemented by *re-use* of existing row type implementations.

The constraint language of RS offers standard equality and subtyping constraints, as well as a form of conditional constraints. To ensure that only meaningful types and constraints can be built, we equip them with *kinds*, defined by:

$$k ::= \mathit{Con} \mid \mathit{Row}(\tau)_A \mid \mathit{Row}(c)_B \mid \mathit{Type}$$

where A ranges over finite subsets of field labels $\mathcal{L}_a$ and B ranges over finite subsets of set urelements $\mathcal{L}_b$. Row kinds are parameterized by τ or c, specifying whether they describe the contents of a record or a set, respectively. For every kind k, we assume given a distinct, denumerable set of *type variables* $\mathcal{V}_k$. We use $\alpha, \beta, \gamma, \ldots$ to represent type variables. From here on, we consider only *well-kinded* types and constraints, as defined in Fig. 3.4 and Fig. 3.5. The purpose of these rules is to guarantee that every constraint has a well-defined interpretation within our model, defined in Sect. 3.2.2.

3.2.2 The model

The model for RS is constructed by associating with every kind k a mathematical structure denoted $[\![k]\!]$. Each of these structures contain elements which can be informally described as *ground* types— that is, type variable-free types— of the relevant kind. We denote these elements $\hat{\tau}$. Each structure $[\![k]\!]$ is equipped with a *partial ordering* $\leq$ of its elements. Accordingly, the ordering on each $[\![k]\!]$ is transitive and reflexive, so the following inferences are axiomatic for all $\hat{\tau}$:

$$\hat{\tau} \leq \hat{\tau} \qquad\qquad \frac{\hat{\tau} \leq \hat{\tau}' \qquad \hat{\tau}' \leq \hat{\tau}''}{\hat{\tau} \leq \hat{\tau}''}$$

The model is explicated for each $[\![k]\!]$ as follows:

$[\![\mathit{Con}]\!]$: The elements of $[\![\mathit{Con}]\!]$ are contained in the set $\{+, -, \bot, \top\}$. As is made clear by the full definition of our model, continued below, the characteristics of the ordering $\leq$ over the model is determined by the definition of $\leq$ over $[\![\mathit{Con}]\!]$; if we define $\leq$ over $[\![\mathit{Con}]\!]$ as equality, then $\leq$ is an equivalence relation over the entire model– that is, over each $[\![k]\!]$. On the other hand, we may choose between two subtype orderings over $[\![\mathit{Con}]\!]$, the first omitting the constructors $\top$ and $\bot$ and axiomatized as $+ \leq -$, the second axiomatized as follows:

$$\bot \leq + \qquad\qquad \bot \leq - \qquad\qquad + \leq \top \qquad\qquad - \leq \top$$

In other words, these orderings generate the following lattices:

$$\begin{tikzcd} - \\ + \arrow[u] \end{tikzcd} \qquad \text{and} \qquad \begin{tikzcd} & \top & \\ + \arrow[ur] & & - \arrow[ul] \\ & \bot \arrow[ul] \arrow[ur] & \end{tikzcd}$$

The second ordering will allow a sound use of conditional constraints in a typing of $?_r$. It would also allow a sound use of conditional constraints to type a fully general difference operation, as in [38]. However, there is no necessity for such an operation in this thesis. By choosing these orderings, we

$$\rho(\tau \text{ ref}) = \rho(\tau) \text{ ref} \qquad \rho(\tau \to \tau') = \rho(\tau) \to \rho(\tau')$$

$$\rho(\{\tau\}) = \{\rho(\tau)\} \qquad \rho(\{\cdot\tau\cdot\}) = \{\cdot\rho(\tau)\cdot\}$$

$$\rho(\ell : \tau \,;\, \tau')(\ell) = \rho(\tau) \qquad \rho(\ell : \tau \,;\, \tau')(\ell') = \rho(\tau')(\ell') \qquad (\ell \neq \ell')$$

$$\rho(\partial\tau)(\ell) = \rho(\tau) \qquad \rho(c) = c$$

Figure 3.6: Type-to-kind assignment definition

generate models of structural, atomic subtyping. Note well that although the symbols $\bot$ and $\top$ are used, the reader should not be misled into thinking that this is a non-structural subtyping system.

$[\![\mathit{Row}(\tau)_A]\!]$ and $[\![\mathit{Row}(c)_B]\!]$: Given a finite set of labels $A \subseteq \mathcal{L}_a$, $[\![\mathit{Row}(\tau)_A]\!]$ is the set of total, almost constant functions from $\mathcal{L}_a \backslash A$ into $[\![\mathit{Type}]\!]$. (A function is *almost constant* if it is constant except on a finite number of inputs.) In short, $\mathit{Row}(\tau)_A$ is the kind of rows which do *not* carry the fields mentioned in A; $\mathit{Row}(\tau)_\varnothing$ is the kind of complete rows. Similarly, $[\![\mathit{Row}(c)_B]\!]$ is the set of total, almost constant functions from $\mathcal{L}_b \backslash B$ into $[\![\mathit{Con}]\!]$, so that $\mathit{Row}(c)_B$ is the kind of set types which do *not* carry the fields mentioned in B, and $\mathit{Row}(c)_\varnothing$ is the kind of complete set types. The ordering $\leq$ is extended inductively to $[\![\mathit{Row}(c)_B]\!]$ and coinductively with $[\![\mathit{Type}]\!]$ to $[\![\mathit{Row}(\tau)_A]\!]$, pointwise and covariantly, as follows:

$$\frac{\hat{\tau}, \hat{\tau}' \in [\![\mathit{Row}(\tau)_A]\!] \qquad \forall a \in \mathcal{L}_a \backslash A \,.\, \hat{\tau}(a) \leq \hat{\tau}'(a)}{\hat{\tau} \leq \hat{\tau}'}$$

$$\frac{\hat{\tau}, \hat{\tau}' \in [\![\mathit{Row}(c)_B]\!] \qquad \forall b \in \mathcal{L}_b \backslash B \,.\, \hat{\tau}(b) \leq \hat{\tau}'(b)}{\hat{\tau} \leq \hat{\tau}'}$$

$[\![\mathit{Type}]\!]$: The elements of $[\![\mathit{Type}]\!]$ are contained in the free algebra generated by the constructors $\to$, with signature $[\![\mathit{Type}]\!] \times [\![\mathit{Type}]\!] \to [\![\mathit{Type}]\!]$, and $\{\cdot\}$, with signature $[\![\mathit{Row}(\tau)_\varnothing]\!] \to [\![\mathit{Type}]\!]$. The ordering $\leq$ is coinductively extended with $[\![\mathit{Row}(\tau)_A]\!]$ to $[\![\mathit{Type}]\!]$ by treating the constructor $\to$ as contravariant in the first argument and covariant in the second, and by treating the constructors $\{\cdot\}$ and $\{\cdot\cdot\cdot\}$ as covariant; that is:

$$\frac{\hat{\tau}_1' \leq \hat{\tau}_1 \qquad \hat{\tau}_2 \leq \hat{\tau}_2'}{\hat{\tau}_1 \to \hat{\tau}_2 \leq \hat{\tau}_1' \to \hat{\tau}_2'} \qquad \frac{\hat{\tau} \leq \hat{\tau}'}{\{\hat{\tau}\} \leq \{\hat{\tau}'\}} \qquad \frac{\hat{\tau} \leq \hat{\tau}'}{\{\cdot\hat{\tau}\cdot\} \leq \{\cdot\hat{\tau}'\cdot\}}$$

This completes the definition of the model.

$$\frac{}{\rho \vdash \textbf{true}} \qquad \frac{\rho \vdash C_1 \qquad \rho \vdash C_2}{\rho \vdash C_1 \wedge C_2} \qquad \frac{\rho = \rho' \,[\alpha] \qquad \rho' \vdash C}{\rho \vdash \exists \alpha . C}$$

$$\frac{\rho(\tau) = \rho(\tau')}{\rho \vdash \tau = \tau'} \qquad \frac{\rho(\tau) \leq \rho(\tau')}{\rho \vdash \tau \leq \tau'} \qquad \frac{c \leq \rho(\tau) \Rightarrow \rho \vdash \tau' \leq \tau''}{\rho \vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''}$$

$$\frac{\tau, \tau', \tau'' : \mathit{Row}(c)_B \qquad \forall b \in \mathcal{L}_b \backslash B \,.\, (c \leq \rho(\tau)(b) \Rightarrow \rho(\tau')(b) \leq \rho(\tau'')(b))}{\rho \vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''}$$

Figure 3.7: Interpretation of constraints

3.2.3 Interpretation in the model

We may now give the interpretation of types and constraints within the model. It is parameterized by an *assignment* ρ, i.e. a function which, for every kind k, maps $\mathcal{V}_k$ into $[\![k]\!]$. The interpretation of types is obtained by extending ρ so as to map every type of kind k to an element of $[\![k]\!]$, as defined in Fig. 3.6. Fig. 3.7 defines the constraint satisfaction predicate $\cdot \vdash \cdot$, whose arguments are an assignment ρ and a constraint C. (The notation $\rho = \rho' \,[\alpha]$ means that ρ and ρ' coincide except possibly on α.) These rules are not particularly surprising, except those that involve conditional constraints of the form if $\tau \leq c$ then $\tau' \leq \tau''$, where τ is a set type; we call these *complex* conditional constraints. The meaning and utility of complex conditional constraints will be demonstrated in subsection 3.3.2. Constraint *entailment* is defined as usual: $C \Vdash C'$ (read: C entails C') holds iff, for every assignment ρ, $\rho \vdash C$ implies $\rho \vdash C'$.

We refer to the type and constraint logic, together with its interpretation, as RS. More precisely, we have defined two logics, where $\leq$ is interpreted as either equality or as one of two non-trivial subtype orderings. We will refer to them as RS$^{=}$, RS$^{\leq_1}$, and RS$^{\leq_2}$, respectively.

3.2.4 Abbreviated set types

Although the set types defined in previous sections are expressive, and the form of their contents as kinds of row types allows re-use of existing implementations, an abbreviation of their form is possible— in fact, we may define the readable, succinct type form presented in Chapter 1 as syntactic sugar for primitive set and row types. Each field $b : \tau$ is shortened to $b\tau$. We also define abbreviated row type constructors $\varnothing$ and ω, specifying that all elements not otherwise mentioned in a row are absent or present, respectively. For example, the set $\{r_1, r_2\}$ will be one (and the only) value of type $\{r_1+, r_2+, \varnothing\}$. Formally, the grammar for abbreviated types is defined as follows:

$$\varsigma \quad ::= \quad \{\varsigma\} \mid b\tau, \varsigma \mid \omega \mid \varnothing \mid \beta \qquad\qquad \textit{abbreviated set types}$$

The interpretation of abbreviated types $(\!|\, \varsigma \,|\!)$ as primitive types is defined as follows:

$$\begin{aligned}
(\!|\, \{\varsigma\} \,|\!) &= \{\cdot (\!|\, \varsigma \,|\!) \cdot\} \\
(\!|\, b\tau, \varsigma \,|\!) &= (b : \tau\, ;\; (\!|\, \varsigma \,|\!)) \\
(\!|\, \varnothing \,|\!) &= \partial - \\
(\!|\, \omega \,|\!) &= \partial + \\
(\!|\, \beta \,|\!) &= \beta
\end{aligned}$$

We say that an abbreviated type ς is well-kinded iff $(\!|\, \varsigma \,|\!)$ is, and we write $\rho(\varsigma)$ to denote $\rho((\!|\, \varsigma \,|\!))$. In the presentation of the type system for pml_B, we will use abbreviated set types for a more succinct and readable presentation; however, we note that their definition as syntactic sugar for primitive types allows for an implementation that re-uses existing row type implementations.

For brevity, we also define the following shorthand for partial rows; however, we note that this is a convenience for this presentation, not a proposed addition to the type machinery per se. Letting $B = \{r_1, \ldots, r_n\}$

$$\begin{aligned}
(Bc) &\triangleq (r_1 c, \cdots, r_n c) \\
(B\bar{c}) &\triangleq (r_1 c_1, \cdots, r_n c_n) \\
(B\bar{\gamma}) &\triangleq (r_1 \gamma_1, \cdots, r_n \gamma_n)
\end{aligned}$$

So for example, $\{B+, \varnothing\} = \{r_1+, \cdots, r_n+, \varnothing\}$. When considering subtyping relations over variable-free terms, we may also use rows themselves (partial or complete) to denote their representatives in the model, e.g. we may assert $R+ \leq R-$ in $\mathrm{RS}^{\leq_1}$; see especially Lemma 3.5.

3.3 Types for pml_B

To define a type system for pml_B, we instantiate HM(X) with one of RS^{rel}, where *rel* ranges over $\{=, \leq\}$, and postulate records, sets, and associated operations, along with their semantics, as extensions of the core HM(X) language. We also define initial type bindings for these extensions, which we prove sound. This obtains a sound type system for pml_B— more than one, in fact, since our choice of rel results in either a unification- or subtyping-based system.

3.3.1 Constants and initial type bindings for pml_B

To begin our conception of pml_B as an extension of the core HM(X) language, we postulate the constant $\{\cdot\}$, and the families of constants $a.\cdot$ and $\cdot\{a = \cdot\}$. The constants of pml_B, along with their initial type bindings, are then specified in Fig. 3.8, with the exception of $?_b$. For this constant, we provide two alternative bindings, defined in Fig. 3.9: the first is less accurate, requiring the branches of the membership test to have an identical type. The second is more expressive, allowing different types in each branch, but requires the use of conditional constraints.

Additionally, we must define initial types for records. This is accomplished with the following definition:

Definition 3.1 *Let $v-$ range over record-free values. Then the initial type bindings of records are inductively defined as follows:*

- $\{v\} : \{\partial\tau\}$, *where* $v : \tau$ *in* $\mathcal{S}_1^{=}$
- $v_1\{a = v_2\} : \{a : \tau_1 \,;\; \partial\tau_2\}$, *where* $v_1 : \{a : \tau_1' \,;\; \partial\tau_2\}$ *and* $v_2 : \tau_1$ *in* $\mathcal{S}_1^{=}$

The initial bindings for records, record operations, set definitions and membership check are easily understood. The bindings for all other set operations contain conditional constraints; their meaning is more subtle, and is discussed in the following section.

3.3.2 Bindings with complex conditional constraints

As is evident in Fig. 3.8, we make extensive use of complex conditional constraints to provide accurate types for set operations. To demonstrate the behavior and usefulness of conditional constraints in application to set operation types, we give the following example. Let the sets B_1 and B_2 be defined as follows:

$$\begin{aligned} B_1 &= \{b_1, b_2, b_3\} \\ B_2 &= \{b_1, b_2, b_4\} \end{aligned}$$

Suppose then that we wish to type the expression $B_1 \wedge B_2$, using the unification-based constraint system RS$^{=}$. Given the typing for $\wedge$ defined in Fig. 3.8, the variables β_1 and β_2 will be unified with the types of the contents of B_1 and B_2, respectively:

$$\begin{aligned}\beta_1 &= (b_1+, b_2+, b_3+, \varnothing)\\ \beta_2 &= (b_1+, b_2+, b_4+, \varnothing)\end{aligned}$$

Additionally, β_3 will be unified with a type that is "splittable" into the appropriate form for the expansion of the complex conditional constraint in the type of $\wedge$:

$$\beta_3 = (b_1\gamma_1, b_2\gamma_2, b_3\gamma_3, b_4\gamma_4, \beta)$$

Then, given the rules for complex conditional constraints defined in Fig. 3.7, the constraint C in the type of $\wedge$ can be expanded as follows:

$$\begin{aligned}C = \quad & \text{if } - \leq + \text{ then } - \leq \gamma_1 \wedge \text{ if } + \leq + \text{ then } + \leq \gamma_1\\ \wedge\ & \text{if } - \leq + \text{ then } - \leq \gamma_2 \wedge \text{ if } + \leq + \text{ then } + \leq \gamma_2\\ \wedge\ & \text{if } - \leq + \text{ then } - \leq \gamma_3 \wedge \text{ if } + \leq + \text{ then } - \leq \gamma_3\\ \wedge\ & \text{if } - \leq - \text{ then } - \leq \gamma_4 \wedge \text{ if } + \leq - \text{ then } + \leq \gamma_4\\ \wedge\ & \text{if } - \leq \varnothing \text{ then } \varnothing \leq \beta \wedge \text{ if } + \leq \varnothing \text{ then } \varnothing \leq \beta\end{aligned}$$

This expansion will force the following unification:

$$\begin{aligned}\beta_3 &= (b_1+, b_2+, b_3-, b_4-, \varnothing)\\ & \textit{or}\\ \beta_3 &= (b_1+, b_2+, \varnothing)\end{aligned}$$

And this in fact is the type of $\{b_1, b_2\}$, and $B_1 \wedge B_2$ evaluates to $\{b_1, b_2\}$.

3.3.3 Type soundness for pml_B

Given the previous development, we may now define the type system for pml_B. In fact, we may succinctly define four distinct type systems for pml_B, with varying degrees of expressiveness:

Definition 3.2 (pml_B **type systems)** *Let Δ_1 (resp. Δ_2) be the initial environment containing all bindings specified in Fig. 3.8, and binding (1) (resp. (2)) for $?_b$ as specified in Fig. 3.9. Then for all $i \in \{1, 2\}$ and $rel \in \{=, \leq_i\}$, the type system $\mathcal{S}_i^{rel}$ is obtained by extending* $\mathrm{HM}(\mathrm{RS}^{rel})$ *with Δ_i as the initial binding environment.*

$$
\begin{array}{rcl}
\{\cdot\} & : & \forall\alpha.\alpha \to \{\partial\alpha\} \\
\cdot\{a = \cdot\} & : & \forall\alpha_1\alpha_2\beta.\{a : \alpha_1\ ;\ \beta\} \to \alpha_2 \to \{a : \alpha_2\ ;\ \beta\} \\
\cdot.a & : & \forall\alpha\beta.\{a : \alpha\ ;\ \beta\} \to \alpha \\
B & : & \{B+, \varnothing\} \\
\bar{B} & : & \{B-, \omega\} \\
\ni_b & : & \forall\beta.\{b+, \beta\} \to \{b+, \beta\} \\
\wedge & : & \forall\beta_1\beta_2\beta_3[C].\{\beta_1\} \to \{\beta_2\} \to \{\beta_3\} \\
 & & \quad \text{where } C = \quad \text{if } - \leq \beta_1 \text{ then } \varnothing \leq \beta_3 \\
 & & \qquad\qquad\qquad \wedge\ \text{if } + \leq \beta_1 \text{ then } \beta_2 \leq \beta_3 \\
\vee & : & \forall\beta_1\beta_2\beta_3[C].\{\beta_1\} \to \{\beta_2\} \to \{\beta_3\} \\
 & & \quad \text{where } C = \quad \text{if } + \leq \beta_1 \text{ then } \omega \leq \beta_3 \\
 & & \qquad\qquad\qquad \wedge\ \text{if } - \leq \beta_1 \text{ then } \beta_2 \leq \beta_3 \\
\ominus & : & \forall\beta_1\beta_2\beta_3[C].\{\beta_1\} \to \{\beta_2\} \to \{\beta_3\} \\
 & & \quad \text{where } C = \quad \text{if } + \leq \beta_1 \text{ then } \varnothing \leq \beta_3 \\
 & & \qquad\qquad\qquad \wedge\ \text{if } - \leq \beta_1 \text{ then } \beta_2 \leq \beta_3
\end{array}
$$

Figure 3.8: Constants and initial type bindings for pml_B

The advantage of using the HM(X) framework for defining the pml_B type systems is now made evident by the proof of their soundness. The only real effort necessary is to prove soundness of the initial type bindings in each Δ_i with respect to the operational semantics of the pml_B language constants, the so called δ-typability property (Definition 2.6). The proof of δ-typability is delayed until the next section, to maintain the flow of this one. Given δ-typability (Lemma 3.5), the soundness of RS^{rel}, and results demonstrated in Chapter 2, we now may immediately obtain type soundness for pml_B in each $\mathcal{S}_i^{rel}$:

Theorem 3.1 **(pml_B Progress)** *If e is a pml_B expression which is well-typed in $\mathcal{S}_i^{rel}$, then e either diverges or reduces to a value.*

Proof. By Definition 3.2, Lemma 3.5, Definition 2.7 and Lemma 2.10. □

$$
\begin{array}{llll}
(1) & ?_b & : & \forall \alpha\beta\gamma.\{b\gamma,\beta\} \to (\{b+,\beta\} \to \alpha) \to (\{b-,\beta\} \to \alpha) \to \alpha \\
(2) & ?_b & : & \forall \bar{\alpha}\bar{\beta}\gamma[C].\{b\gamma,\beta\} \to (\{b+,\beta_1\} \to \alpha_1) \to (\{b-,\beta_2\} \to \alpha_2) \to \alpha \\
 & & & \qquad \text{where } C = \quad \text{if } + \leq \gamma \text{ then } \beta \leq \beta_1 \wedge \text{ if } - \leq \gamma \text{ then } \beta \leq \beta_2 \\
 & & & \qquad\qquad\qquad\quad \wedge \text{ if } + \leq \gamma \text{ then } \alpha_1 \leq \alpha \wedge \text{ if } - \leq \gamma \text{ then } \alpha_2 \leq \alpha
\end{array}
$$

Figure 3.9: Binding options for $?_b$

Theorem 3.2 (pml_B **Type Safety**) *If e is a pml_B expression which is well-typed in $\mathcal{S}_i^{rel}$, then e does not go wrong.*

Proof. Immediate by Theorem 3.1. $\square$

A consequence of Theorem 3.2 is that certain pml_B runtime optimizations may be effected. For example, this result implies that all membership checks $\ni_b$ may be removed at runtime from a well-typed program. This property is verified by the following result, which follows by type soundness:

Corollary 3.1 *Let $\rightsquigarrow$ be defined as $\to$, but with the memcheck rule redefined as $B \ni b \rightsquigarrow B$; that is, no runtime membership checks are performed. Suppose e is well typed; then $e \rightsquigarrow^\star v$ iff $e \to^\star v$.*

3.3.4 δ-typability for pml_B

The statement of δ-typability (Definition 2.6) requires that each functional language constant be δ-defined as a function of one argument, so to state the δ rules for $\vee$, $\wedge$ and $\ominus$ we posit the *subprimitives* $\vee_B$, $\wedge_B$ $\vee_{\bar{B}}$ and $\ominus_B$. The δ rules for pml_B are then defined in Fig. 3.10, as are the subprimitive type bindings which we add to each Δ_i. Note that this construction is made solely for our conception of pml_B as an instance of HM(X); subprimitives are called such because they are not made visible to the programmer.

We begin by stating some Lemmas that describe unsurprising properties of pml_B type judgements for records and sets.

Lemma 3.1 *Let v be a closed pml_B value. If $C, \Gamma \vdash v : \{a_1 : \tau_1 ;\ \tau'\}$ holds in $\mathcal{S}_i^{rel}$, then v is a record of the form $\{v_1\}$ or $v'\{a_1 = v_1\} \cdots \{a_n = v_n\}$ and $C, \Gamma \vdash v_1 : \tau_1$.*

$$
\begin{array}{rcll}
\delta(\{\cdot\}, v) & = & \{v\} & \\
\delta(\cdot\{a = \cdot\}, v) & = & \lambda x.v\{a = x\} & \\
\delta(\cdot.a, \{v\}) & = & v & \\
\delta(\cdot.a, v_1\{a = v_2\}) & = & v_2 & \\
\delta(\cdot.a, v_1\{a' = v_2\}) & = & \delta(\cdot.a, v) & a \neq a' \\
\delta(\vee, B) & = & \vee_B & \\
\delta(\vee, \bar{B}) & = & \vee_{\bar{B}} & \\
\delta(\wedge, B) & = & \wedge_B & \\
\delta(\wedge, \bar{B}) & = & \ominus_B & \\
\delta(\ominus, B) & = & \ominus_B & \\
\delta(\ominus, \bar{B}) & = & \wedge_B & \\
\delta(\vee_{B_1}, B_2) & = & B_1 \cup B_2 & \\
\delta(\vee_{\bar{B}_1}, B_2) & = & \bar{B}_1 \cup B_2 & \\
\delta(\wedge_{B_1}, B_2) & = & B_1 \cap B_2 & \\
\delta(\ominus_{B_1}, B_2) & = & B_2 \backslash B_1 & \\
\delta(?_b, B) & = & \lambda f.\lambda g.f(B) & b \in B \\
\delta(?_b, B) & = & \lambda f.\lambda g.g(B) & b \notin B
\end{array}
$$

$$
\begin{array}{|lcl|}
\hline
\vee_B & : & \forall\beta\bar{\gamma}.\{B\bar{\gamma}, \beta\} \to \{B+, \beta\} \\
\vee_{\bar{B}} & : & \forall\beta\bar{\gamma}.\{B\bar{\gamma}, \beta\} \to \{B\bar{\gamma}, \omega\} \\
\wedge_B & : & \forall\beta\bar{\gamma}.\{B\bar{\gamma}, \beta\} \to \{B\bar{\gamma}, \varnothing\} \\
\ominus_B & : & \forall\bar{\gamma}\beta.\{B\bar{\gamma}, \beta\} \to \{B-, \beta\} \\
\hline
\end{array}
$$

Figure 3.10: δ rules and subprimitive type bindings for pml_B

Lemma 3.2 *Let v be a closed pml_B value. If $C, \Gamma \vdash v : \tau$ holds in $\mathcal{S}_i^{rel}$ and $C \Vdash \tau \leq \{\varsigma\}$ for some set type ς, then v is a set B and $C \Vdash \{B+, \varnothing\} \leq \tau$ or v is a coset $\bar{B}$ and $C \Vdash \{B-, \omega\} \leq \tau$.*

Lemma 3.3 *If $C, \Gamma \vdash B : \{B'+, \tau\}$ holds in $\mathcal{S}_i^{rel}$ then $B' \subseteq B$.*

The following Lemma allows us to characterize the results of set unions, intersections and differences, and like the previous Lemmas, will be useful for proving δ-typability.

Lemma 3.4 *Let s be either a set B or a coset $\bar{B}$; then if $C, \Gamma \vdash s : \{B'\bar{c}, \tau\}$ holds in $\mathcal{S}_i^{rel}$ then so does $C, \Gamma \vdash s \cup B' : \{B'+, \tau\}$ and $C, \Gamma \vdash s \cap B' : \{B'\bar{c}, \varnothing\}$ and $C, \Gamma \vdash s \backslash B' : \{B'-, \tau\}$.*

Now, we may prove the central result of this section, by case analysis on language constants, including subprimitives.

Lemma 3.5 pml_B *is δ-typable in $\mathcal{S}_i^{rel}$.*

Proof. Since Δ_i is similar for each of $\mathcal{S}_i^{rel}$, we can prove δ-typability with respect to each system in a similar manner. Suppose $C, \Gamma \vdash \mathbf{c} : \tau_1 \to \tau_2$ and $C, \Gamma \vdash v : \tau_1$. Then by Lemma 2.6 and definition of each Δ_i, for all cases of $\mathbf{c}$ except $?_b$ in $\mathcal{S}_2^{\leq}$ (to be treated separately), we have a subderivation of the following form in each system $\mathcal{S}_i^{rel}$, where $\varphi = [\bar{\tau}/\bar{\alpha}]$:

$$\dfrac{\dfrac{\dfrac{\Delta_i(\mathbf{c}) = \forall\bar{\alpha}.\tau_1' \to \tau_2'}{C, \Gamma \vdash \mathbf{c} : \forall\bar{\alpha}.\tau_1' \to \tau_2'}}{C, \Gamma \vdash \mathbf{c} : \varphi(\tau_1') \to \varphi(\tau_2')} \qquad C \Vdash \varphi(\tau_1') \to \varphi(\tau_2') \leq \tau_1 \to \tau_2}{C, \Gamma \vdash \mathbf{c} : \tau_1 \to \tau_2}$$

To show that $\delta(\mathbf{c}, v)$ is defined and $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2$, we proceed by case analysis on $\mathbf{c}$:

Case $\cdot.a_1$. In this case $\varphi = [\tau/\alpha, \tau'/\beta]$ so that $\varphi(\tau_1') \to \varphi(\tau_2') = \{a_1 : \tau\,;\ \tau'\} \to \tau$, hence $\tau_1 = \{a_1 : \tau''\,;\ \tau'''\}$ such that $C \Vdash \tau'' \leq \tau$, and also $C \Vdash \tau \leq \tau_2$, by properties of $\leq$. But since $C, \Gamma \vdash v : \tau_1$ by assumption, therefore v is a record of the form $\{v_1\}$ or $v'\{a_1 = v_1\} \cdots \{a_n = v_n\}$ and $C, \Gamma \vdash v_1 : \tau''$, by Lemma 3.1. Thus $\delta(\cdot.a_1, v) = v_1$ by definition, and since $C \Vdash \tau'' \leq \tau_2$ by transitivity of $\leq$, we have $C, \Gamma \vdash \delta(\cdot.a_1, v) : \tau_2$ in this case by SUB.

Cases $\{\cdot\}$ and $\cdot\{a = \cdot\}$ follow by definition of δ_i.

Case $\ni_b$. In this case $\varphi = [\tau/\beta]$ so that $\varphi(\tau_1') \to \varphi(\tau_2') = \{b+, \tau\} \to \{b+, \tau\}$, hence $C \Vdash \tau_1 \leq \{b+, \tau\}$ and $C \Vdash \{b+, \tau\} \leq \tau_2$ by properties of $\leq$. But then by Lemma 3.2 we have that v is a set B, and since $C, \Gamma \vdash v : \{b+, \tau\}$ by assumption and SUB, by Lemma 3.3 we have that $\{b\} \subseteq B$, hence $\delta(\ni_b, v) = v$ by definition. Thus $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_1$ by assumption, and $C \Vdash \tau_1 \leq \tau_2$ by transitivity of $\leq$, hence $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2$ in this case by SUB.

Case $\vee_B$. In this case $\varphi(\tau_1')$ is a set type by definition of each Δ_i, therefore by properties of $\leq$ and Lemma 3.2 we have that v is a set B', so that $\delta(\vee_B, B') = B \cup B'$ by definition. Thus, we have $\varphi = [\bar{c}/\bar{\gamma}, \tau/\beta]$ and $\varphi(\tau_1') \to \varphi(\tau_2') = \{B\bar{c}, \tau\} \to \{B+, \tau\}$, where $C \Vdash \tau_1 \leq \{B\bar{c}, \tau\}$ and $C \Vdash \{B+, \tau\} \leq \tau_2$ by assumption and properties of $\leq$. But then $C, \Gamma \vdash B' : \{B\bar{c}, \tau\}$ by SUB, so $C, \Gamma \vdash B' \cup B : \{B+, \tau\}$ by Lemma 3.4, so $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2$ in this case by SUB.

Case $\vee_{\bar{B}}$. In this case $\varphi(\tau_1')$ is a set type by definition of each Δ_i, therefore by properties of $\leq$ and Lemma 3.2 we have that v is a set B', so that $\delta(\vee_{\bar{B}}, B') = \bar{B} \cup B'$ by definition. Let $B_1 = B \cap B'$ and $B_2 = B \backslash B_1$ and $B_3 = B' \backslash B_2$. Then B_1, B_2 and B_3 are mutually disjoint and $B = B_1 \cup B_2$ and $B' = B_1 \cup B_3$. Thus, we have that $\varphi(\tau_1') \to \varphi(\tau_2') = \{B_1\bar{c_1}, B_2\bar{c_2}, B_3\bar{c_3}, \tau\} \to \{B_1\bar{c_1}, B_2\bar{c_2}, B_3+, \omega\}$ by definition of $\Delta_i(\vee_{\bar{B}})$. But since $C, \Gamma \vdash B' : \tau_1$ by assumption, and

$C \Vdash \tau_1 \leq \varphi(\tau_1')$ by properties of $\leq$, and $C \Vdash \{B_1+, B_2-, B_3+, \varnothing\} \leq \tau_1$ by Lemma 3.2, therefore $B_1+ \leq B_1\bar{c_1}$ and $B_2- \leq B_2\bar{c_2}$ by properties of $\leq$. Further, since $C, \Gamma \vdash \bar{B} : \{B_1-, B_2-, B_3+, \omega\}$ by definition of Δ_i, therefore $C, \Gamma \vdash \bar{B} \cup B' : \{B_1+, B_2-, B_3+, \omega\}$ by Lemma 3.4. But then $C, \Gamma \vdash \bar{B} \cup B' : \{B_1\bar{c_1}, B_2\bar{c_2}, B_3+, \omega\}$ by the above reasoning and SUB, and $C \Vdash \varphi(\tau_2') \leq \tau_2$ by properties of $\leq$, therefore $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2$ in this case by SUB.

Case $\wedge_B$. In this case $\varphi(\tau_1')$ is a set type by definition of each Δ_i, therefore by properties of $\leq$ and Lemma 3.2 we have that v is a set B', so that $\delta(\wedge_B, B') = B \cap B'$ by definition. Thus, we have $\varphi = [\bar{c}/\bar{\gamma}, \tau/\beta]$ and $\varphi(\tau_1') \to \varphi(\tau_2') = \{B\bar{c}, \tau\} \to \{B\bar{c}, \varnothing\}$, where $C \Vdash \tau_1 \leq \{B\bar{c}, \tau\}$ and $C \Vdash \{B\bar{c}, \varnothing\} \leq \tau_2$ by assumption and properties of $\leq$. But then $C, \Gamma \vdash B' : \{B\bar{c}, \tau\}$ by SUB, so that $C, \Gamma \vdash B \cap B' : \{B\bar{c}, \varnothing\}$ by Lemma 3.4, hence $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2$ in this case by SUB.

Case $\ominus_B$. In this case $\varphi(\tau_1')$ is a set type by definition of each Δ_i, therefore by properties of $\leq$ and Lemma 3.2 we have that v is a set B', so that $\delta(\ominus_B, B') = B' \backslash B$ by definition. Thus, we have $\varphi = [\bar{c}/\bar{\gamma}, \tau/\beta]$ and $\varphi(\tau_1') \to \varphi(\tau_2') = \{B\bar{c}, \tau\} \to \{B-, \tau\}$, where $C \Vdash \tau_1 \leq \{B\bar{c}, \tau\}$ and $C \Vdash \{B-, \tau\} \leq \tau_2$ by assumption and properties of $\leq$. But then $C, \Gamma \vdash B' : \{B\bar{c}, \tau\}$ by SUB, so $C, \Gamma \vdash B' \backslash B : \{B-, \tau\}$ by Lemma 3.4, so $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2$ in this case by SUB.

Case $\vee$. In this case τ_1 is a set type by definition of Δ_i and properties of $\forall$ ELIM and SUB. Then by Lemma 3.2 we have that v is either a set B or a coset $\bar{B}$, so the proof proceeds by subcases:

Subcase $v = B$. In this subcase $\tau_1 = \{B\bar{c}, \varsigma\}$ where $B+ \leq B\bar{c}$ (considering subtyping in the model for brevity) and $\varnothing \leq \varsigma$ by Lemma 3.2. Also, $\varphi(\tau_1') \to \varphi(\tau_2')$ is of the form $\{\varsigma_1\} \to \{\varsigma_2\} \to \{\varsigma_3\}$ by definition of Δ_i, where $\varsigma_1 = (B\bar{c'}, \varsigma')$ such that $B\bar{c} \leq B\bar{c'}$ and $\varsigma \leq \varsigma'$ by properties of $\leq$. By definition of $\Delta_i(\vee)$ and interpretation of conditional constraints we have $\varsigma_2 = (B\bar{c_2}, \varsigma_2')$ and $\varsigma_3 = (B\bar{c_3}, \varsigma_3')$ such that $B+ \leq B\bar{c_3}$, since $B+ \leq B\bar{c} \leq B\bar{c'}$, and $\varsigma_2' \leq \varsigma_3'$, since $\varnothing \leq \varsigma \leq \varsigma'$. But then $\tau_2 = \{B\bar{c_4}, \varsigma_4\} \to \{B\bar{c_5}, \varsigma_5\}$ where $B\bar{c_4} \leq B\bar{c_2}$ and $\varsigma_4 \leq \varsigma_2'$ by properties of $\leq$, and also $B\bar{c_3} \leq B\bar{c_5}$ and $\varsigma_3' \leq \varsigma_5$, again by properties of $\leq$. Now, since $v = B$ in this subcase, therefore $\delta(\vee, B) = \vee_B$, and by $\forall$ ELIM we have $C, \Gamma \vdash \vee_B : \{B\bar{c_4}, \varsigma_4\} \to \{B+, \varsigma_4\}$. But by the above reasoning we have $\varsigma_4 \leq \varsigma_2' \leq \varsigma_3' \leq \varsigma_5$, and also $B+ \leq B\bar{c_3} \leq B\bar{c_5}$, therefore $C, \Gamma \vdash \vee_B : \{B\bar{c_4}, \varsigma_4\} \to \{B\bar{c_5}, \varsigma_5\}$, i.e. $C, \Gamma \vdash \vee_B : \tau_2$, so this subcase holds. Subcase $v = \bar{B}$ follows in a similar manner.

Case $\wedge$. In this case τ_1 is a set type by definition of Δ_i and properties of $\forall$ ELIM and SUB. Then by Lemma 3.2 we have that v is either a set B or a coset $\bar{B}$, so the proof proceeds by subcases:

Subcase $v = B$. In this subcase $\tau_1 = \{B\bar{c}, \varsigma\}$ where $B+ \leq B\bar{c}$ (considering subtyping in the model for brevity) and $\varnothing \leq \varsigma$ by Lemma 3.2. Also, $\varphi(\tau_1') \to \varphi(\tau_2')$ is of the form $\{\varsigma_1\} \to \{\varsigma_2\} \to \{\varsigma_3\}$ by definition of Δ_i, where $\varsigma_1 = (B\bar{c'}, \varsigma')$ such that $B\bar{c} \leq B\bar{c'}$ and $\varsigma \leq \varsigma'$ by properties of $\leq$. By definition of $\Delta_i(\wedge)$ and interpretation of conditional constraints we have $\varsigma_2 = (B\bar{c_2}, \varsigma_2')$ and $\varsigma_3 = (B\bar{c_3}, \varsigma_3')$ such that $B\bar{c_2} \leq B\bar{c_3}$, since $B+ \leq B\bar{c} \leq B\bar{c'}$, and $\varnothing \leq \varsigma_3'$, since $\varnothing \leq \varsigma \leq \varsigma'$. But then $\tau_2 = \{B\bar{c_4}, \varsigma_4\} \to \{B\bar{c_5}, \varsigma_5\}$ where $B\bar{c_4} \leq B\bar{c_2}$ and $\varsigma_4 \leq \varsigma_2'$ by properties of $\leq$, and also $B\bar{c_3} \leq B\bar{c_5}$ and $\varsigma_3' \leq \varsigma_5$, again by properties of $\leq$. Now, since $v = B$ in this subcase, therefore $\delta(\wedge, B) = \wedge_B$, and by $\forall$ ELIM we have $C, \Gamma \vdash \vee_B : \{B\bar{c_4}, \varsigma_4\} \to \{B\bar{c_4}, \varnothing\}$. But by the above reasoning we have $\varnothing \leq \varsigma_3' \leq \varsigma_5$, and also $B\bar{c_4} \leq B\bar{c_2} \leq B\bar{c_3} \leq B\bar{c_5}$, therefore $C, \Gamma \vdash \vee_B : \{B\bar{c_4}, \varsigma_4\} \to \{B\bar{c_5}, \varsigma_5\}$, i.e. $C, \Gamma \vdash \vee_B : \tau_2$, so this subcase holds. Subcase $v = \bar{B}$ follows in a similar manner.

Case $\ominus$. In this case τ_1 is a set type by definition of Δ_i and properties of $\forall$ ELIM and SUB. Then by Lemma 3.2 we have that v is either a set B or a coset $\bar{B}$, so the proof proceeds by subcases:

Subcase $v = B$. In this subcase $\tau_1 = \{B\bar{c}, \varsigma\}$ where $B+ \leq B\bar{c}$ (considering subtyping in the model for brevity) and $\varnothing \leq \varsigma$ by Lemma 3.2. Also, $\varphi(\tau_1') \to \varphi(\tau_2')$ is of the form $\{\varsigma_1\} \to \{\varsigma_2\} \to \{\varsigma_3\}$ by definition of Δ_i, where $\varsigma_1 = (B\bar{c'}, \varsigma')$ such that $B\bar{c} \leq B\bar{c'}$ and $\varsigma \leq \varsigma'$ by properties of $\leq$. By definition of $\Delta_i(\ominus)$ and interpretation of conditional constraints we have $\varsigma_2 = (B\bar{c_2}, \varsigma_2')$ and $\varsigma_3 = (B\bar{c_3}, \varsigma_3')$ such that $B- \leq B\bar{c_3}$, since $B+ \leq B\bar{c} \leq B\bar{c'}$, and $\varsigma_2' \leq \varsigma_3'$, since $\varnothing \leq \varsigma \leq \varsigma'$. But then $\tau_2 = \{B\bar{c_4}, \varsigma_4\} \to \{B\bar{c_5}, \varsigma_5\}$ where $B\bar{c_4} \leq B\bar{c_2}$ and $\varsigma_4 \leq \varsigma_2'$ by properties of $\leq$, and also $B\bar{c_3} \leq B\bar{c_5}$ and $\varsigma_3' \leq \varsigma_5$, again by properties of $\leq$. Now, since $v = B$ in this subcase, therefore $\delta(\ominus, B) = \ominus_B$, and by $\forall$ ELIM we have $C, \Gamma \vdash \ominus_B : \{B\bar{c_4}, \varsigma_4\} \to \{B-, \varsigma_4\}$. But by the above reasoning we have $\varsigma_4 \leq \varsigma_2' \leq \varsigma_3' \leq \varsigma_5$, and also $B- \leq B\bar{c_3} \leq B\bar{c_5}$, therefore $C, \Gamma \vdash \ominus_B : \{B\bar{c_4}, \varsigma_4\} \to \{B\bar{c_5}, \varsigma_5\}$, i.e. $C, \Gamma \vdash \ominus_B : \tau_2$, so this subcase holds. Subcase $v = \bar{B}$ follows in a similar manner.

Case $?_b$, subcase $\mathcal{S}_1^=$. In this case $\varphi(\tau_1')$ is a set type by definition of Δ_1, therefore by Lemma 3.2 we have that v is a set B and $C \Vdash \tau_1 = \{B+, \varnothing\}$, so $C, \Gamma \vdash B : \{B+, \varnothing\}$ by SUB. Let $B' = B - \{r\}$; then $C \Vdash \tau_2 = (\{b+, B'+, \delta-\} \to \tau) \to (\{b-, B'+, \delta-\} \to \tau) \to \tau$ for some τ by definition of Δ_1. But then $\delta(\mathbf{c}, v) = \lambda f.\lambda g.f(B)$ in this case if $b \in B$, and $\delta(\mathbf{c}, v) = \lambda f.\lambda g.g(B)$ in this case if $b \notin B$. Suppose on the one hand that $b \in B$; then $C, \Gamma \vdash \lambda f.\lambda g.f(B) : (\{B+, \delta-\} \to \tau) \to (\{b-, B'+, \delta-\} \to \tau) \to \tau$ is derivable by the type of B specified above, two applications of ABS and an instance of APP, so $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2$ holds in this case by SUB. The result follows in a similar manner if $b \notin B$. Subcase $\mathcal{S}_1^{\leq}$ follows analogously modulo subtyping coercions.

Case $?_b$, subcase $\mathcal{S}_2^i$. Let:

$$\begin{aligned}
\forall\bar{\alpha}\bar{\beta}\gamma[D].\tau &= \Delta_2(?_b) \\
\varphi &= [\tau_\beta/\beta, \tau_{\beta_1}/\beta_1, \tau_{\beta_2}/\beta_2, \tau_\alpha/\alpha, \tau_{\alpha_1}/\alpha_1, \tau_{\alpha_2}/\alpha_2, c/\gamma] \\
\tau_1' &= \{bc, \tau_\beta\} \\
\tau_2' &= (\{b+, \tau_{\beta_1}\} \to \tau_{\alpha_1}) \to (\{b-, \tau_{\beta_2}\} \to \tau_{\alpha_2}) \to \tau_\alpha
\end{aligned}$$

Then by Lemma 2.6 and definition of Δ_2 we have the following derivation in this case:

$$\dfrac{\dfrac{\dfrac{\Delta_2(?_b) = \forall\bar{\alpha}\bar{\beta}\gamma[D].\tau}{C, \Gamma \vdash ?_b : \forall\bar{\alpha}\bar{\beta}\gamma[D].\tau} \quad C \Vdash \varphi(D)}{C, \Gamma \vdash ?_b : \tau_1' \to \tau_2'} \qquad C \Vdash \tau_1' \to \tau_2' \leq \tau_1 \to \tau_2}{C, \Gamma \vdash ?_b : \tau_1 \to \tau_2}$$

By properties of $\leq$ we have that $C \Vdash \tau_1 \leq \tau_1'$, so by Lemma 3.2 we have that v is a set B. and $C \Vdash \{B+, \varnothing\} \leq \tau_1$. Suppose on the one hand that $b \in B$; then since $C \Vdash \{B+, \varnothing\} \leq \tau_1'$ by transitivity of $\leq$, it must be the case that $C \Vdash + \leq c$ by definition of τ_1' and $\leq$.

Now, since we've assumed $b \in B$, therefore $\delta(?_b, B) = \lambda f.\lambda g.f(B)$ by definition. Let $\tau_1'' = \{b+, \tau_\beta\} \to \tau_\alpha$ and $\tau_2'' = \{b-, \tau_{\beta_2}\} \to \tau_{\alpha_2}$. Then since $C, \Gamma \vdash B : \tau_1$ holds by assumption and $C \Vdash \tau_1 \leq \tau_1'$ by properties of $\leq$, therefore $C, \Gamma \vdash B : \tau_1'$ by SUB. Hence by Lemma 3.4, $C, \Gamma \vdash B : \{b+, \tau_{\beta_1}\}$ is derivable, so the following derivation is valid in $\mathcal{S}_2^{\leq}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{(\Gamma; f : \tau_1''; g : \tau_2'')(f) = \tau_1''}{C, (\Gamma; f : \tau_1''; g : \tau_2'') \vdash f : \tau_1''} \quad C, \Gamma \vdash B : \{b-, \tau_\beta\}}{C, (\Gamma; f : \tau_1''; g : \tau_2'') \vdash f(B) : \tau_\alpha}}{C, (\Gamma; f : \tau_1'') \vdash \lambda g.f(B) : \tau_2'' \to \alpha}}{C, \Gamma \vdash \lambda f.\lambda g.f(B) : \tau_1'' \leq \tau_2'' \to \alpha}$$

But $C \Vdash \tau_\beta \leq \tau_{\beta_1} \wedge \tau_{\alpha_1} \leq \tau_\alpha$ holds since $C \Vdash \varphi(D)$ and $C \Vdash + \leq c$ by the above, therefore by properties of $\leq$ we have that $C \Vdash \tau_1'' \to \tau_2'' \to \tau_\alpha \leq \tau_2'$, and also $C \Vdash \tau_2' \leq \tau_2$, so by transitivity of $\leq$ and SUB we have $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2$ in this case. If $b \notin B$ the case follows in a similar manner. □

Although the pml_B subprimitives are not visible to the programmer, it is possible to imagine them as syntactic sugar for partial application of the general set operations, e.g. $\wedge_B \triangleq \wedge B$; the following result demonstrates that the types remain consistent. This result will be useful in the next Chapter.

Lemma 3.6 *The following types are valid in any $\mathcal{S}_i^{rel}$:*

$$\begin{aligned}
\vee B &: \forall \beta\bar{\gamma}.\{B\bar{\gamma},\beta\} \to \{B+,\beta\} \\
\wedge B &: \forall \beta\bar{\gamma}.\{B\bar{\gamma},\beta\} \to \{B\bar{\gamma},\varnothing\} \\
\ominus B &: \forall \beta\bar{\gamma}.\{B\bar{\gamma},\beta\} \to \{B\bar{-},\beta\}
\end{aligned}$$

Proof. We consider the type of $\vee B$ first. Let φ be defined as follows:

$$\varphi \;\triangleq\; [\,(B+,\varnothing)/\beta_1,\;\; (B\bar{\gamma},\beta)/\beta_2,\;\; (B+,\beta)/\beta_3\,]$$

Then, given $\Delta_i(\vee)$ as defined in Fig. 3.8, with constraint C:

$$\begin{aligned}
\varphi(C) \;=\; & \quad \text{if } + \leq (B+,\varnothing) \text{ then } (B+,\varnothing) \leq (B+,\beta) \\
& \wedge \text{ if } - \leq (B+,\varnothing) \text{ then } (B\bar{\gamma},\beta) \leq (B+,\beta)
\end{aligned}$$

Now, assuming that B contains n elements and given the interpretation of constraints defined in Fig. 3.7, this constraint is equivalent to the following:

$$\begin{aligned}
\varphi(C) \;=\; & \quad (\text{if } + \leq + \text{ then } + \leq +) \wedge \cdots \wedge\; (\text{if } + \leq + \text{ then } + \leq +) \\
& \wedge (\text{if } + \leq \varnothing \text{ then } \varnothing \leq \beta) \\
& \wedge (\text{if } - \leq + \text{ then } \gamma_1 \leq +) \wedge \cdots \wedge\; (\text{if } - \leq + \text{ then } \gamma_n \leq +) \\
& \wedge (\text{if } + \leq \beta \text{ then } \beta \leq \beta)
\end{aligned}$$

We may then assert that **true** $\Vdash \varphi(C)$: clearly, each constraint $(\text{if } + \leq + \text{ then } + \leq +)$ holds for any assignment, as does $(\text{if } + \leq \varnothing \text{ then } \varnothing \leq \beta)$ vacuously. Furthermore, each constraint of the form $(\text{if } - \leq + \text{ then } \gamma_i \leq +)$ holds vacuously for any assignment; and, for any assignment of β the constraint $\beta \leq \beta$ must hold, hence the constraint $(\text{if } + \leq \beta \text{ then } \beta \leq \beta)$ holds for any assignment. Thus **true** $\Vdash \varphi(C)$ is valid, so by rules CONST and $\forall$ ELIM we have the following derivation:

$$\frac{\textbf{true}, \varnothing \vdash \vee : \Delta_i(\vee) \qquad \textbf{true} \Vdash \varphi(C)}{\textbf{true}, \varnothing \vdash \vee : \{B+,\varnothing\} \to \{B\bar{\gamma},\beta\} \to \{B+,\beta\}}$$

Furthermore, by rules CONST and APP, and since **true** $=$ **true** $\wedge$ **true** we have:

$$\frac{\textbf{true} \wedge \textbf{true}, \varnothing \vdash \vee : \{B+,\varnothing\} \to \{B\bar{\gamma},\beta\} \to \{B+,\beta\} \qquad \textbf{true} \wedge \textbf{true}, \varnothing \vdash B : \{B+,\varnothing\}}{\textbf{true} \wedge \textbf{true}, \varnothing \vdash \vee B : \{B\bar{\gamma},\beta\} \to \{B+,\beta\}}$$

Finally, by $\forall$ INTRO we have:

$$\frac{\mathbf{true} \wedge \mathbf{true}, \varnothing \vdash \vee B : \{B\bar{\gamma}, \beta\} \to \{B+, \beta\} \qquad \beta\bar{\gamma} \cap \mathrm{fv}(\mathbf{true} \wedge \mathbf{true}, \varnothing) = \varnothing}{\mathbf{true} \wedge \exists \beta\bar{\gamma}.\mathbf{true}, \varnothing \vdash \vee B : \forall \beta\bar{\gamma}[\mathbf{true}].\{B\bar{\gamma}, \beta\} \to \{B+, \beta\}}$$

Therefore, since $\mathbf{true} = \mathbf{true} \wedge \exists \beta\bar{\gamma}.\mathbf{true}$ and:

$$\forall \beta\bar{\gamma}[\mathbf{true}].\{B\bar{\gamma}, \beta\} \to \{B+, \beta\} \triangleq \forall \beta\bar{\gamma}.\{B\bar{\gamma}, \beta\} \to \{B+, \beta\}$$

the Lemma holds in this case. The Lemma with respect to $\wedge B$ and $B\ominus$ follows in a similar manner.

□

3.3.5 Type inference

Recall from Sect. 2.4 that for any instance of HM(X), it is sufficient to define gen and $normalize$ functions that satisfy the specifications laid out in Definitions 2.12 and 2.14 to obtain type inference. For the systems $\mathcal{S}_i^{rel}$, the definition of the gen function is trivial. To define $normalize$, it is sufficient to turn to previous work to obtain a satisfactory procedure.

In [31], a method of row type unification is defined. In [24], a subtyping constraint solution algorithm is defined that treats conditional constraints. Thus, the HM(X) inference algorithm can be instantiated with the method described in [31] to obtain λ_{sec} type inference in $\mathcal{S}_1^=$. The algorithm can be instantiated with the method described in [24] to obtain λ_{sec} type inference in $\mathcal{S}_i^{\leq}$, and $\mathcal{S}_2^=$ with subtyping interpreted as equality. Since these methods have been proven correct in the cited texts in a manner that complies with Definition 2.14, correct type inference for the systems $\mathcal{S}_i^{rel}$ is an immediate consequence– another manifest benefit of our use of HM(X).

Using our OCaml implementation of the HM(X) type inference algorithm in the appendix, the $\mathcal{S}_1^=$ type inference algorithm may be implemented via the module:

```
Hm = Hmx.Make(GroundSig)(System)(Primitives)
```

where `GroundSig` is an implementation of core HM(X) types, and `System` is a unification-based constraint system with solution algorithm `unify`, with:

```
constrain node term = unify node term
```

The module `Primitives` is an implementation of the initial type bindings in Δ_1 defined in Sect. 3.3.1. The definitions of `GroundSig`, `System` and `Primitives` are not included in the Appendix, but are available online at `http://www.cs.jhu.edu/~ces/thesis/impl/indirect`.

Chapter 4

Types for Access Control, Revisited

In this chapter, we return to our consideration of a static approach to stack inspection-style security, first discussed in Chapter 1. The $\lambda_{\text{sec}}^{\mathsf{S}}$ language introduced in that chapter is here extended and recast into a more technically appealing form, called λ_{sec}. In particular, we do away with explicit stacks, using appropriately defined evaluation contexts to implicitly represent security information.

The type system for λ_{sec}, including soundness results, is obtained via transformation into the pml_B language that preserves the meaning of programs. This approach allows a proof of soundness for the λ_{sec} type system to be derived from type soundness in pml_B, conserving significant proof effort compared to an *ab initio* approach. This approach also allows re-use of the set types presented in Chapter 3, including existing implementations.

4.1 The λ_{sec} language definition

We now formally define the λ_{sec} language. Syntactically, the language is much the same as $\lambda_{\text{sec}}^{\mathsf{S}}$, but is simplified and extended with some new constructs. The semantics is presented in a significantly different manner, without any explicit stack definitions; however, we demonstrate that the language presented in Chapter 1 may be embedded in the redefined language, showing that this simplified specification accurately reflects the JDK 1.2 implementation, since it subsumes $\lambda_{\text{sec}}^{\mathsf{S}}$.

4.1.1 Syntax

The λ_{sec} grammar is defined in Fig. 4.1. Stacks and related constructs, i.e. function closures and framed expressions, are unnecessary. Following Fournet and Gordon [9], we redefine

$$
\begin{array}{lr}
r \in \mathcal{R}, R \subseteq \mathcal{R} & \textit{resources} \\
p \in \mathcal{P}, P \subseteq \mathcal{P}, \textit{where } \mathcal{P} = 2^{\mathcal{R}} & \textit{principals} \\
v ::= x \mid \mathsf{fix}\, z.\lambda x.f & \textit{values} \\
e ::= v \mid e\, e \mid \mathsf{let}\, x = e\, \mathsf{in}\, e \mid \mathsf{enable}\, r\, \mathsf{in}\, e \mid \mathsf{check}\, r\, \mathsf{then}\, e \mid & \textit{expressions} \\
\qquad \mathsf{test}\, r\, \mathsf{then}\, e\, \mathsf{else}\, e \mid f & \\
f ::= p.e & \textit{signed expressions} \\
E ::= [] \mid E\, e \mid v\, E \mid \mathsf{let}\, x = E\, \mathsf{in}\, e \mid \mathsf{enable}\, r\, \mathsf{in}\, E \mid p.E & \textit{evaluation contexts}
\end{array}
$$

Figure 4.1: Grammar for λ_{sec}

the set of *principals* $\mathcal{P}$ as the powerset of $\mathcal{R}$— that is, we identify a principal with the set of resources to which it has access. We use p and P to range over principals and over sets thereof, respectively, and write *nobody* for the empty privilege set, that is, for the principal with no access rights.

As in the version of HM(X) presented in Chapter 2, function definitions now contain a recursive binding mechanism, and we write $\lambda x.f$ to denote $\mathsf{fix}\, z.\lambda x.f$ when z has no free occurrences in f. We also add a let construct to the language, to set the stage for let-polymorphism.

In the λ_{sec} definition, we replace the dopriv_r constant with expressions of the form $\mathsf{enable}\, r\, \mathsf{in}\, e$. Recalling that dopriv_r is used in application to functions with a dummy argument, expressions $\mathsf{enable}\, r\, \mathsf{in}\, e$ activate r for the evaluation of e, and are thus a simplification of dopriv_r applications. We also define a new construct that allows privilege *testing*— that is, branching on the presence or absence of a particular privilege: expressions of the form $\mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ evaluate to e_1 if r is active, and e_2 otherwise. This mechanism allows λ_{sec} to reflect a common idiom in the Java JDK1.2 implementation, where an exception resulting from a failed privilege check is caught and handled. As we will see in Sect. 4.4, using the typing machinery developed in the two previous chapters allows us to treat this extension in a precise, yet flexible, manner.

Evaluation contexts for λ_{sec} are also defined, and include ownership and privilege activation information. This addition to evaluation contexts allows a redefinition of stack inspection that infers stack-based security information from evaluation contexts.

$$\frac{r \in p \qquad S \vdash r}{S.p \vdash r} \qquad \frac{S \vdash r}{S.r' \vdash r} \qquad \frac{S \vdash r^+}{S.r \vdash r} \qquad \frac{S \vdash r^+}{S.r' \vdash r^+} \qquad \frac{r \in p}{S.p \vdash r^+}$$

Figure 4.2: Backward stack inspection algorithm

$$\frac{\mathit{nobody}, \varnothing, S \vdash R \quad r \in R}{S \vdash r} \qquad p, R, \epsilon \vdash R \qquad \frac{q, R \cap q, S \vdash R'}{p, R, q.S \vdash R'} \qquad \frac{p, R \cup (\{r\} \cap p), S \vdash R'}{p, R, r.S \vdash R'}$$

Figure 4.3: Forward stack inspection algorithm

4.1.2 Stack inspection

We now give a simplified specification of the stack inspection process, by noticing that stacks are implicitly contained in evaluation contexts, whose grammar is defined in Fig. 4.1. Indeed, a context defines a path from the term's root down to its active redex, along which one finds exactly the security annotations which the JDK 1.2 would maintain on the stack, that is, code owners p and enabled resources r. In Sect. 4.2, we will demonstrate that the explicit stack inspection semantics may be simulated in the implicit semantics.

To formalize this idea, we associate to every evaluation context E a finite string $|E|$ of principals and resources, called a *stack*. The right-most letters in the string correspond to the most recent stack frames. We write ϵ for the empty stack and $S_1.S_2$ for the concatenation of the stacks S_1 and S_2.

$$\begin{array}{rclrcl} |[]| &=& \epsilon & |E\,e| &=& |E| \\ |v\,E| &=& |E| & |\mathsf{let}\,x = E\,\mathsf{in}\,e| &=& |E| \\ |\mathsf{enable}\,r\,\mathsf{in}\,E| &=& r.|E| & |p.E| &=& p.|E| \end{array}$$

We can now define a new, "implicit stack" inspection algorithm. We give two variants of it, a backward (Fig. 4.2) and a forward one (Fig. 4.3). The former scans the stack, starting with the most recent frames, then moving toward their ancestors. The latter, on the other hand, scans the stack in the order it was built. Furthermore, its formulation is altered so that it internally computes not only whether access to a given resource r is legal, but also the set of all resources which may be legally accessed given the current stack. These algorithms are referred to as *lazy* and *eager*, respectively, by Gong [11]. While the former is employed by most current JVM implementations, the latter forms the basis of the security-passing style [45] translation which we will introduce in Sect. 4.3.

The following theorem states that forward and backward stack inspection are in fact equivalent. Subsequently, we will write $S \vdash r$ without specifying which of the two algorithms is being used. We will also write $E \vdash r$ for $|E| \vdash r$.

Theorem 4.1 *Assume given a stack S and a resource r. Let P stand for the set of all principals that contain r. Then, the following three statements are equivalent:*

1. *$S \vdash r$ holds according to the rules of Fig. 4.2;*
2. *$S \vdash r$ holds according to the rules of Fig. 4.3;*
3. *some suffix of S belongs to the regular language $P\mathcal{R}^\star r(P \mid \mathcal{R})^\star$.*

Proof. We begin by proving that the first statement is equivalent to the third one. First, check that the auxiliary judgement $S \vdash r^+$ holds if and only if some suffix of S belongs to $P\mathcal{R}^\star$. Then, check that $S \vdash r$ holds, according to the rules of Fig. 4.2, if and only if some suffix of S belongs to the regular language $P\mathcal{R}^\star r(P \mid \mathcal{R})^\star$. Each of these checks is immediate.

We now prove that the second statement is equivalent to the third one. Let A (resp. B, resp. C) be the set of stacks S such that $\exists R' \ni r \quad p, R, S \vdash R'$ for some (or, equivalently, for all) p, R such that $p \not\ni r \wedge R \not\ni r$ (resp. $p \ni r \wedge R \not\ni r$, resp. $p \ni r \wedge R \ni r$). It is straightforward to check that, according to the last three rules in Fig. 4.3, A, B and C are the least solutions to the following recursive equations:

$$\begin{aligned}
A &::= P.B \mid (\mathcal{P} \setminus P).A \mid \mathcal{R}.A \\
B &::= P.B \mid (\mathcal{P} \setminus P).A \mid r.C \mid (\mathcal{R} \setminus \{r\}).B \\
C &::= \epsilon \mid (\mathcal{P} \setminus P).A \mid (P \mid \mathcal{R}).C
\end{aligned}$$

An inductive argument shows that $A \subseteq B \subseteq C$ holds. Then, through a few rewriting steps, one can bring the equations into a form where it is evident that A is exactly $(\mathcal{P} \mid \mathcal{R})^\star P\mathcal{R}^\star r(P \mid \mathcal{R})^\star$. We do not give the details. In principle, the check can be mechanized by verifying that the minimal deterministic finite automaton (over the 4-symbol alphabet $\{r\}$, $\mathcal{R} \setminus \{r\}$, P and $\mathcal{P} \setminus P$) associated with this regular expression is exactly the one described by the above equations. There remains to conclude by noticing that, according to the first rule in Fig. 4.3, $S \vdash r$ holds if and only if $S \in A$.

□

4.1.3 Operational semantics

We define the operational semantics of λ_{sec}, using implicit stack inspection, via the following reduction rules:

$$\begin{array}{rcll}
E[(\mathsf{fix}\, z.\lambda x.f)\, v] & \to & E[f[v/x][\mathsf{fix}\, z.\lambda x.f/z]] & \\
E[\mathsf{let}\, x = v\, \mathsf{in}\, e] & \to & E[e[v/x]] & \\
E[\mathsf{check}\, r\, \mathsf{then}\, e] & \to & E[e] & \text{if } E \vdash r \\
E[\mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2] & \to & E[e_1] & \text{if } E \vdash r \\
E[\mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2] & \to & E[e_2] & \text{if } \neg(E \vdash r) \\
E[\mathsf{enable}\, r\, \mathsf{in}\, v] & \to & E[v] & \\
E[p.v] & \to & E[v] &
\end{array}$$

An evaluation context E is a component of every rule, which allows inspection of it when needing to perform security checks. Note that it is *not* the case that $e \to e'$ implies $E[e] \to E[e']$. Indeed, enclosing e within a new evaluation context E may cause more or fewer privileges to be enabled, changing the outcome of stack inspection.

The first two rules are standard. The next rule allows $\mathsf{check}\, r\, \mathsf{then}\, e$ to reduce into e only if stack inspection succeeds (as expressed by the side condition $E \vdash r$); otherwise, execution is blocked. The following two rules use stack inspection in a similar way to determine how to reduce $\mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$; however, they never cause execution to fail. The last two rules state that security annotations become unnecessary once the expression they enclose has been reduced to a value. In the explicit stack inspection semantics of Chapter 1, these rules are implemented simply by popping stack frames (and the security annotations they contain) after executing a method.

This operational semantics constitutes a concise, formal description of Java stack inspection in a higher-order setting. It is easy to check that every closed term either is a value, or is reducible, or is of the form $E[\mathsf{check}\, r\, \mathsf{then}\, e]$ where $\neg(E \vdash r)$. Terms of the third category are *stuck*; they represent access control violations. An expression e is said to *go wrong* if and only if $e \to^{\star} e'$, where e' is a stuck expression, holds.

4.2 Simulating $\lambda_{\text{sec}}^{\mathsf{S}}$ in λ_{sec}

In this section we demonstrate that $\lambda_{\text{sec}}^{\mathsf{S}}$ can be simulated in λ_{sec}. This result provides confidence in the faithfulness of λ_{sec} to real implementations, since $\lambda_{\text{sec}}^{\mathsf{S}}$ was defined as a low-level model of the JDK1.2 implementation.

We begin by defining transformations from $\lambda_{\text{sec}}^{\mathtt{S}}$ expressions and evaluation contexts to λ_{sec} expressions and contexts. The transformation is fairly obvious, the principal novelty being the treatment of dopriv expressions, which ensures that the evaluation of $[\![e]\!]$ in the transformation of $\mathsf{dopriv}_r e$ is not accorded excessive privileges.

Definition 4.1 *Let ok $\triangleq \lambda x.nobody.x$; then we define the $\lambda_{\text{sec}}^{\mathtt{S}}$-to-$\lambda_{\text{sec}}$ expression transformation as follows:*

$$
\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![p.e]\!] & = & p.[\![e]\!] \\
[\![\lambda x.e]\!] & = & \lambda x.[\![e]\!] \\
[\![\mathtt{C}(\gamma, \lambda x.e)]\!] & = & [\![(\lambda x.e)\gamma]\!] \\
[\![e_1 e_2]\!] & = & [\![e_1]\!][\![e_2]\!] \\
[\![\mathsf{dopriv}_r e]\!] & = & \mathsf{let}\, f_{\text{run}} = [\![e]\!] \,\mathsf{in}\, \mathsf{enable}\, r \,\mathsf{in}\, f_{\text{run}}(ok) \\
[\![\mathsf{check}\, r \,\mathsf{then}\, e]\!] & = & \mathsf{check}\, r \,\mathsf{then}\, [\![e]\!]
\end{array}
$$

Definition 4.2 *We define the $\lambda_{\text{sec}}^{\mathtt{S}}$-to-$\lambda_{\text{sec}}$ context transformation as follows:*

$$
\begin{array}{rcl}
[\![(\mathtt{S}, \gamma, [\,])]\!] & = & [\,] \\
[\![(\mathtt{S}, \gamma, Ee)]\!] & = & [\![(\mathtt{S}, \gamma, E)]\!][\![e\gamma]\!] \\
[\![(\mathtt{S}, \gamma, vE)]\!] & = & [\![v\gamma]\!][\![(\mathtt{S}, \gamma, E)]\!] \\
[\![(\mathtt{S}, \gamma, \mathsf{dopriv}_r E)]\!] & = & \mathsf{let}\, f_{\text{run}} = [\![(\mathtt{S}, \gamma, E)]\!] \,\mathsf{in}\, \mathsf{enable}\, r \,\mathsf{in}\, f_{\text{run}}(ok) \\
[\![(\langle \gamma', p, \mathbf{off} \rangle :: \mathtt{S}, \gamma, \cdot E \cdot)]\!] & = & p.[\![(\mathtt{S}, (\gamma; \gamma'), E)]\!] \\
[\![(\langle \gamma', p, \mathbf{on}(r) \rangle :: \mathtt{S}, \gamma, \cdot E \cdot)]\!] & = & \mathsf{enable}\, r \,\mathsf{in}\, p.[\![(\mathtt{S}, (\gamma; \gamma'), E)]\!]
\end{array}
$$

In the context transformation, stacks are analyzed from the "outside-in". Since stacks are LIFO data structures, this means that the "oldest" variable bindings will be at the bottom of the stack. Thus, while the context transformation deconstructs stacks in the usual manner, we will apply the transformation to reversed stacks; see e.g. Definition 4.3. For this purpose, we define the function rev, where:

$$
\mathrm{rev}(\langle \gamma_1, p_1, \mathbf{f}_1 \rangle :: \cdots :: \langle \gamma_n, p_n, \mathbf{f}_n \rangle :: nil) = (\langle \gamma_n, p_n, \mathbf{f}_n \rangle :: \cdots :: \langle \gamma_1, p_1, \mathbf{f}_1 \rangle :: nil)
$$

Now, we define a *simulation* relation $\lhd$, based on the transformations defined above. It will be our task to demonstrate that this relation is preserved between an arbitrary computation in $\lambda_{\text{sec}}^{\mathtt{S}}$ and its simulacrum in λ_{sec}.

Definition 4.3 *The relation* $(\mathtt{S}, e) \lhd e'$ *holds iff* $\mathtt{S}, e$ *is well-formed, and there exists* E_1 *and* e_1 *such that* $e = E_1[e_1]$ *and* $[\![(\mathrm{rev}(\mathtt{S}), \varnothing, E_1)]\!][[\![\mathtt{S}(e_1)]\!]] = e'$.

Note that this definition allows, to a degree, an arbitrary choice of E_1 and e_1. This raises the question, can a $\lambda_{\mathrm{sec}}^{\mathtt{S}}$ configurations be simulated by more than one λ_{sec} expression? The answer is no, the relation $\lhd$ is in fact a mapping from $\lambda_{\mathrm{sec}}^{\mathtt{S}}$ configurations to λ_{sec} expressions. However, we must prove this fact; we begin by proving some useful properties of the context transformation.

Lemma 4.1 *The following properties hold:*

1. *If* $[\![E[e]]\!]$ *is defined, the frame depth of* E *is* 0
2. *If* $[\![\mathtt{S}(E[e])]\!]$ *is defined, then* $[\![\mathtt{S}(E[e])]\!] = [\![(nil, \mathrm{env}(\mathtt{S}), E)]\!][[\![\mathtt{S}(e)]\!]]$
3. *If* $E = E_1[E_2]$ *where the frame depth of* E_1 *equals the length of* $\mathtt{S}$*, and* $[\![(\mathrm{rev}(\mathtt{S}), \gamma, E)]\!]$ *is defined, then* $[\![(\mathrm{rev}(\mathtt{S}), \gamma, E)]\!] = [\![(\mathrm{rev}(\mathtt{S}), \gamma, E_1)]\!][[\![(nil, (\gamma; \mathrm{env}(\mathtt{S})), E_2)]\!]]$

Proof. Each assertion is treated individually:

1. Immediate by definition of $[\![E[e]]\!]$, since the transformation is defined only on unframed expressions.

2. By structural induction on E. In the basis $E = [\,]$, and since $[\![(nil, \mathrm{env}(\mathtt{S}), [\,])]\!] = [\,]$, therefore $[\![E[e]]\!] = [\![(nil, \mathrm{env}(\mathtt{S}), E)]\!][[\![e]\!]] = [\![e]\!]$ in this case. The proof then proceeds by case analysis on composite E, which excludes contexts of the form $\cdot E' \cdot$ by assertion 1.

Case $E = E'e'$. In this case $[\![(nil, \mathrm{env}(\mathtt{S}), E)]\!] = [\![(nil, \mathrm{env}(\mathtt{S}), E')]\!][\![e'(\mathrm{env}(\mathtt{S}))]\!]$, and since $E[e] = (E'[e])e'$ we also have $[\![\mathtt{S}(E[e])]\!] = [\![\mathtt{S}(E'[e])]\!][\![\mathtt{S}(e')]\!]$. But by the induction hypothesis it is the case that $[\![\mathtt{S}(E'[e])]\!] = [\![(nil, \mathrm{env}(\mathtt{S}), E')]\!][[\![\mathtt{S}(e)]\!]]$, and $[\![(nil, \mathrm{env}(\mathtt{S}), E)]\!][[\![\mathtt{S}(e)]\!]] = [\![(nil, \mathrm{env}(\mathtt{S}), E')]\!][[\![\mathtt{S}(e)]\!]][\![e'(\mathrm{env}(\mathtt{S}))]\!]$, and since $e'(\mathrm{env}(\mathtt{S})) = \mathtt{S}(e')$ and $e(\mathrm{env}(\mathtt{S})) = \mathtt{S}(e)$ by definition, this case holds.

The other cases follow in a similar, straightforward manner by the induction hypothesis, due to the tight correspondence of the term and context transformations.

3. By structural induction on E_1. In the basis we have that $E_1 = [\,]$, so $E_1[E_2] = E_2 = E$, and $\mathtt{S} = nil$ since the frame depth of E_1 equals the length of $\mathtt{S}$ by assumption, so $\mathrm{env}(\mathtt{S}) = \varnothing$ and $\gamma; \mathrm{env}(\mathtt{S}) = \gamma$. But then $[\![(\mathrm{rev}(\mathtt{S}), \gamma, E)]\!] = [\![(nil, \gamma, E)]\!]$, and since $[\![(\mathrm{rev}(\mathtt{S}), \gamma, [\,])]\!] = [\,]$ therefore $[\![(\mathrm{rev}(\mathtt{S}), \gamma, E_1)]\!][[\![(nil, (\gamma; \mathrm{env}(\mathtt{S})), E_2)]\!]] = [\![(nil, \gamma, E_2)]\!]$, so this case hold. The induction step proceeds by case analysis on composite E_1.

Case $E_1 = \cdot E_1' \cdot$, subcase $\mathrm{rev}(\mathtt{S}) = \langle \gamma', p, \mathbf{off} \rangle :: \mathtt{S}'$. In this case we have $E = \cdot E'[E_2] \cdot$ and:

$$\begin{aligned} [\![(\langle \gamma', p, \mathbf{off} \rangle :: \mathtt{S}', \gamma, E_1)]\!] &= p.[\![(\mathtt{S}', (\gamma; \gamma'), E_1')]\!] \\ [\![(\langle \gamma', p, \mathbf{off} \rangle :: \mathtt{S}', \gamma, E)]\!] &= p.[\![(\mathtt{S}', (\gamma; \gamma'), E_1'[E_2])]\!] \end{aligned}$$

by definition and assumption. Let $\mathtt{S}'' = \mathrm{rev}(\mathtt{S}')$; now, since the frame depth of E_1 is equal to the length of $\mathtt{S}$ by assumption, therefore the frame depth of E_1' is equal to the length of $\mathtt{S}'$, so by the induction hypothesis we have that:

$$[\![(\mathtt{S}', (\gamma; \gamma'), E_1')]\!][[\![(\mathit{nil}, (\gamma; \gamma'; \mathrm{env}(\mathtt{S}'')), E_2)]\!]] = [\![(\mathtt{S}', (\gamma; \gamma'), E_1'[E_2])]\!]$$

But since $\mathtt{S}'' = \mathrm{rev}(\mathtt{S}')$, and $\mathrm{rev}(\mathtt{S}) = \langle \gamma', p, \mathbf{off} \rangle :: \mathtt{S}'$, therefore $(\gamma; \gamma'; \mathrm{env}(\mathtt{S}'')) = \gamma; \mathrm{env}(\mathtt{S})$, so we have:

$$[\![(\mathtt{S}', (\gamma; \gamma'), E_1')]\!][[\![(\mathit{nil}, (\gamma; \mathrm{env}(\mathtt{S})), E_2)]\!]] = [\![(\mathtt{S}', (\gamma; \gamma'), E_1'[E_2])]\!]$$

which gives us:

$$p.[\![(\mathtt{S}', (\gamma; \gamma'), E_1')]\!][[\![(\mathit{nil}, (\gamma; \mathrm{env}(\mathtt{S})), E_2)]\!]] = p.[\![(\mathtt{S}', (\gamma; \gamma'), E_1'[E_2])]\!]$$

therefore this case holds. The other cases follow in a similar manner by the induction hypothesis.

$\square$

Now, we may prove the desired result, that the relation $\lhd$ is indeed a mapping; this will have the advantage of allowing us to *construct* simulations, and be certain that this construction is exhaustive.

Lemma 4.2 *The relation $\lhd$ is a mapping from $\lambda_{\mathrm{sec}}^{\mathtt{S}}$ configurations to λ_{sec} expressions; i.e., if $(\mathtt{S}, e) \lhd e'$ and $(\mathtt{S}, e) \lhd e''$ then $e' = e''$.*

Proof. Let E_1, E_1', e_1 and e_1' be such that $E_1[e_1] = E_1'[e_1'] = e$, and $[\![(\mathrm{rev}(\mathtt{S}), \varnothing, E_1)]\!][[\![\mathtt{S}(e_1)]\!]] = e'$ and $[\![(\mathrm{rev}(\mathtt{S}), \varnothing, E_1')]\!][[\![\mathtt{S}(e_1')]\!]] = e''$. Assume w.l.o.g. that $e_1 = E[e_1']$ for some E, so that $E_1' = E_1[E]$. Since $[\![\mathtt{S}(e_1)]\!]$ is defined, therefore the frame depth of E is 0 by Lemma 4.1, so also by Lemma 4.1 we have that $[\![\mathtt{S}(e_1)]\!] = [\![(\mathit{nil}, \mathrm{env}(\mathtt{S}), E)]\!][[\![\mathtt{S}(e_1')]\!]]$. Further, since $\mathtt{S}, e$ is well-formed by definition of $\lhd$ and $[\![\mathtt{S}(e_1)]\!]$ is defined and thus e_1 is unframed, therefore the frame depth of E_1 equals the length of $\mathtt{S}$, so that $[\![(\mathrm{rev}(\mathtt{S}), \varnothing, E_1')]\!] = [\![(\mathrm{rev}(\mathtt{S}), \varnothing, E_1)]\!][[\![(\mathit{nil}, \mathrm{env}(\mathtt{S}), E)]\!]]$ by Lemma 4.1. Thus $[\![(\mathrm{rev}(\mathtt{S}), \varnothing, E_1')]\!][[\![\mathtt{S}(e_1')]\!]] = [\![(\mathrm{rev}(\mathtt{S}), \varnothing, E_1)]\!][[\![(\mathit{nil}, \mathrm{env}(\mathtt{S}), E)]\!]][[\![\mathtt{S}(e_1')]\!]] = [\![(\mathrm{rev}(\mathtt{S}), \varnothing, E_1)]\!][[\![\mathtt{S}(e_1)]\!]]$, so $e' = e''$. $\square$

Next, we want to demonstrate that the transformation preserves security information with respect to stack inspection in $\lambda_{\text{sec}}^{\mathtt{S}}$ and λ_{sec}. This is accomplished with the following Lemmas, followed by other miscellaneous results.

Lemma 4.3 *Let* $|\mathtt{S}|$ *be defined as follows:*

$$\begin{array}{rcl} |nil| & = & \epsilon \\ |\langle\gamma, p, \mathbf{off}\rangle :: \mathtt{S}| & = & |\mathtt{S}|.p \\ |\langle\gamma, p, \mathbf{on}(r)\rangle :: \mathtt{S}| & = & |\mathtt{S}|.p.r \end{array}$$

Then it is the case that $\mathrm{inspect}(\mathtt{S}, r) = \textbf{true}$ *iff* $|\mathtt{S}| \vdash r$.

Proof. Straightforward by definition of $\mathrm{inspect}$, the backward stack inspection algorithm presented in this Chapter, and induction on the length of the stack. □

Lemma 4.4 *If* $[\![(\mathrm{rev}(\mathtt{S}), \gamma, E)]\!] = E'$*, then* $|\mathtt{S}| = |E'|$.

Proof. Straightforward by definition of the translation and induction on E. □

Lemma 4.5 *If* $[\![(\mathrm{rev}(\mathtt{S}), \gamma, E_1)]\!] = E_2$ *then* $\mathrm{inspect}(\mathtt{S}, r) = \textbf{true}$ *iff* $E_2 \vdash r$.

Proof. Immediate by Lemma 4.3 and Lemma 4.4. □

Lemma 4.6 $[\![e]\!][[\![v]\!]/x] = [\![e[v/x]]\!]$

Proof. By structural induction on e. In the basis $e = x'$; since $[\![x']\!] = x'$, if $x' \neq x$ then we have $[\![e]\!][[\![v]\!]/x] = [\![e[v/x]]\!] = x'$, otherwise $[\![e]\!][[\![v]\!]/x] = [\![e[v/x]]\!] = [\![v]\!]$. The other cases follow trivially by the induction hypothesis. □

Lemma 4.7 *The following assertions hold:*

1. *If* $[\![(\mathrm{rev}(\mathtt{S}), \gamma, E_1)]\!] = E_2$ *then* $[\![(\mathrm{rev}(\langle\gamma', p, \mathbf{off}\rangle :: \mathtt{S}), \gamma, E_1[\cdot[\,]\cdot])]\!] = E_2[p.[\,]]$.
2. *If* $[\![(\mathrm{rev}(\mathtt{S}), \gamma, E_1)]\!] = E_2$ *then* $[\![(\mathrm{rev}(\langle\gamma', p, \mathbf{on}(r)\rangle :: \mathtt{S}), \gamma, E_1[\cdot[\,]\cdot])]\!] = E_2[\mathsf{enable}\, r\, \mathsf{in}\, p.[\,]]$.

Proof. Both assertions follow by a straightforward structural induction on E_1. □

Now, we may demonstrate a simulation result with regard to one step of reduction in $\lambda_{\text{sec}}^{\mathtt{S}}$, stated and proved as follows:

Lemma 4.8 *If* $(\mathtt{S}, e_e) \lhd e_i$ *and* $\mathtt{S}, e_e \to \mathtt{S}', e'_e$*, then* $e_i \to^\star e'_i$ *such that* $(\mathtt{S}', e'_e) \lhd e'_i$*.*

Proof. By definition, any reduction $\mathtt{S}, e_e \to \mathtt{S}', e'_e$ can be taken as an instance of *context*, where $e_e = E_1[e_1]$ and $e'_e = E_1[e'_1]$ and $\mathtt{S}, e_1 \to \mathtt{S}', e'_1$ by a reduction rule other than context. The proof then proceeds by case analysis on these rules:

Case *var*. In this case we have $e_1 = x$, $e'_1 = \mathtt{S}(x)$ and $\mathtt{S} = \mathtt{S}'$. Let $e_2 = [\![\mathtt{S}(x)]\!]$ and $E_2 = [\![(\mathrm{rev}(\mathtt{S}), \varnothing, E_1)]\!]$, so that $(\mathtt{S}, e_e) \lhd E_2[e_2]$ by definition. But since $\mathrm{rng}(\mathrm{env}(\mathtt{S}))$ is a set of closed values by definition, therefore $e_2 = [\![\mathtt{S}(\mathtt{S}(x))]\!]$, hence $(\mathtt{S}, e'_e) \lhd E_2[e_2]$ by definition, and $E_2[e_2] \to^\star E_2[e_2]$ by reflexivity of $\to^\star$.

Case *closure*. In this case we have $e_1 = \lambda x.e$, $e'_1 = \mathtt{C}(\mathrm{env}(\mathtt{S}), \lambda x.e)$ and $\mathtt{S}' = \mathtt{S}$. Let $E_2 = [\![(\mathrm{rev}(\mathtt{S}), \varnothing, E_1)]\!]$ and $e_2 = [\![(\lambda x.e)(\mathrm{env}(\mathtt{S}))]\!]$, so that $(\mathtt{S}, e_e) \lhd E_2[e_2]$ by definition. Further, by definition of the transformation we have:

$$[\![\mathtt{C}(\mathrm{env}(\mathtt{S}), \lambda x.e)]\!] = [\![(\lambda x.e)(\mathrm{env}(\mathtt{S}))]\!]$$

therefore $(\mathtt{S}, e'_e) \lhd E_2[e_2]$, and $E_2[e_2] \to^\star E_2[e_2]$ by reflexivity of $\to^\star$, so this case holds.

Case *app*. In this case we have $e_1 = \mathtt{C}(\gamma, \lambda x.p.e)v$, $e'_1 = \cdot e \cdot$ and $\mathtt{S}' = \langle \gamma[v/x], p, \mathbf{off} \rangle :: \mathtt{S}$. Let $E_2 = [\![(\mathrm{rev}(\mathtt{S}), \varnothing, E_1)]\!]$ and $e_2 = [\![\mathtt{C}(\gamma, \lambda x.p.e)v]\!]$ so that $(\mathtt{S}, e_e) \lhd E_2[e_2]$ by definition. Then by definition of the transformation we have:

$$[\![\mathtt{C}(\gamma, \lambda x.p.e)v]\!] = (\lambda x.p.[\![e(\gamma \backslash x)]\!])[\![v]\!]$$

and by definition of λ_{sec} reduction we have:

$$E_2[(\lambda x.p.[\![e(\gamma \backslash x)]\!])[\![v]\!]] \to E_2[p.[\![e(\gamma \backslash x)]\!][[\![v]\!]/x]]$$

But by Lemma 4.6 we have $[\![e(\gamma \backslash x)]\!][[\![v]\!]/x] = [\![e(\gamma \backslash x)[v/x]]\!]$, and clearly $[\![e(\gamma \backslash x)[v/x]]\!] = [\![e(\gamma[v/x])]\!]$; further, since $\mathtt{C}(\gamma, \lambda x.p.e)$ is closed by well-formedness of configurations, therefore $e(\gamma[v/x]) = \mathtt{S}'(e)$, so that $[\![e(\gamma[v/x])]\!] = [\![\mathtt{S}'(e)]\!]$, hence:

$$E_2[p.[\![e(\gamma \backslash x)]\!][[\![v]\!]/x]] = E_2[p.[[\![\mathtt{S}'(e)]\!]]]$$

and since by Lemma 4.7 we have that $[\![(\mathrm{rev}(\mathtt{S}'), \varnothing, E_1[\cdot[\,]\cdot])]\!] = E_2[p.[\,]]$, therefore it is the case that $(\mathtt{S}', e'_e) \lhd E_2[p.[[\![\mathtt{S}'(e)]\!]]]$, so this case holds.

Case *dopriv*. In this case $e_1 = \mathsf{dopriv}_r\, \mathtt{C}(\gamma, \lambda\_.p.e)$, $e'_1 = \cdot e \cdot$ and $\mathtt{S}' = \langle \gamma, p, \mathbf{on}(r) \rangle :: \mathtt{S}$. Let $E_2 = [\![(\mathrm{rev}(\mathtt{S}), \varnothing, E_1)]\!]$ and $e_2 = [\![\mathsf{dopriv}_r\, \mathtt{C}(\gamma, \lambda\_.p.e)]\!]$, so that $(\mathtt{S}, e_e) \lhd E_2[e_2]$ by definition. Then by definition of the transformation we have:

$$[\![\mathsf{dopriv}_r\, \mathtt{C}(\gamma, \lambda\_.p.e)]\!] = (\mathsf{let}\, f_{\mathrm{run}} = \lambda\_.p.[\![e\gamma]\!]\; \mathsf{in\; enable}\, r\; \mathsf{in}\; f_{\mathrm{run}}(ok))$$

and by definition of λ_{sec} reduction we have:

$$E_2[\mathsf{let}\, f_{\text{run}} = \lambda\_.p.[\![e\gamma]\!]\, \mathsf{in}\, \mathsf{enable}\, r\, \mathsf{in}\, f_{\text{run}}(ok)]$$
$$\to^\star$$
$$E_2[\mathsf{enable}\, r\, \mathsf{in}\, ((\lambda\_.p.[\![e\gamma]\!])\, ok)]$$
$$\to$$
$$E_2[\mathsf{enable}\, r\, \mathsf{in}\, (p.[\![e\gamma]\!])]$$

But by Lemma 4.7 we have $[\![(\text{rev}(\mathtt{S}'), \varnothing, E_1[\cdot[\,]\cdot])]\!] = E_2[\mathsf{enable}\, r\, \mathsf{in}\, p.[\,]]$, so this case holds by definition of $\lhd$.

Case *checkpriv*. In this case we have $e_1 = \mathsf{check}\, r\, \mathsf{then}\, e$, $e_1' = e$ and $\mathtt{S}' = \mathtt{S}$, with $\text{inspect}(\mathtt{S}, r) = \mathbf{true}$. Let $E_2 = [\![(\text{rev}(\mathtt{S}), \varnothing, E_1)]\!]$ and $e_2 = [\![\mathsf{check}\, r\, \mathsf{then}\, e]\!]$, so that $(\mathtt{S}, e_e) \lhd E_2[e_2]$ by definition. Then by definition of the transformation we have:

$$[\![\mathsf{check}\, r\, \mathsf{then}\, e]\!] = \mathsf{check}\, r\, \mathsf{then}\, [\![e]\!]$$

and since $\text{inspect}(\mathtt{S}, r) = \mathbf{true}$, therefore $E_2 \vdash r$ by Lemma 4.5, so by definition of λ_{sec} reduction we have:

$$E_2[\mathsf{check}\, r\, \mathsf{then}\, [\![e]\!]] \to E_2[[\![e]\!]]$$

so this case holds by definition of $\lhd$.

Case *pop*. In this case we have $e_1 = \cdot v \cdot$, $e_1' = v$ and $\mathtt{S} = \langle \gamma, p, \mathbf{f}\rangle :: \mathtt{S}'$. Let $e_2 = [\![v]\!]$, and let $E_2 = [\![(\text{rev}(\mathtt{S}'), \varnothing, E_1)]\!]$, so that $(e_i, \mathtt{S}') \lhd E_2[e_2]$ by definition. The proof then proceeds via the following subcases:

Subcase $\mathbf{f} = \mathbf{off}$. In this subcase $[\![(\text{rev}(\mathtt{S}), \varnothing, E_1[\cdot[\,]\cdot])]\!] = E_2[p.[\,]]$ by Lemma 4.7, so that $(\mathtt{S}, e_e) \lhd E_2[p.[\![v]\!]]$ by definition, and by definition of λ_{sec} reduction we have:

$$E_2[p.[\![v]\!]] \to E_2[[\![v]\!]]$$

so this case holds.

Subcase $\mathbf{f} = \mathbf{on}(r)$. In this subcase $[\![(\text{rev}(\mathtt{S}), \varnothing, E_1[\cdot[\,]\cdot])]\!] = E_2[\mathsf{enable}\, r\, \mathsf{in}\, p.[\,]]$ by Lemma 4.7, so that $(\mathtt{S}, e_e) \lhd E_2[\mathsf{enable}\, r\, \mathsf{in}\, p.[\![v]\!]]$ by definition, and by definition of λ_{sec} reduction we have:

$$E_2[\mathsf{enable}\, r\, \mathsf{in}\, p.[\![v]\!]] \to^\star E_2[[\![v]\!]]$$

so this case holds. □

We then extend the previous Lemma to arbitrary computations in $\lambda_{\text{sec}}^{\mathtt{S}}$ in Lemma 4.9. We also demonstrate that the simulation preserves dynamic properties of expressions in Lemma 4.10.

Lemma 4.9 *If* $(\mathtt{S}, e_e) \lhd e_i$ *and* $\mathtt{S}, e_e \to^\star \mathtt{S}', e'_e$, *then* $e_i \to^\star e'_i$ *such that* $(\mathtt{S}', e'_e) \lhd e'_i$.

Proof. By Lemma 4.8 and induction on the length of the reduction $\mathtt{S}, e_e \to^\star \mathtt{S}', e'_e$. □

Lemma 4.10 *If* $(\mathtt{S}, e_e) \lhd e_i$ *and* e_e *is a value, then so is* e_i, *if* e_e *is not a value nor of the form* $\lambda x.e$ *then* e_i *is not a value, and if* $(\mathtt{S}, e_e)$ *is stuck then so is* e_i.

Proof. If e_e is a value then it is a closure, so $e_i = [\![e_e]\!]$ by definition of $\lhd$, and the transformation of a closure is a function, which is of course a value.

If e_e is not a value nor of the form $\lambda x.e$ then e_i is not a value by definition of $\lhd$, since only closures and expressions $\lambda x.e$ are translated to values.

If $(\mathtt{S}, e_e)$ is stuck then $e_e = E[\mathsf{check}\, r\, \mathsf{then}\, e]$ and $\mathrm{inspect}(S, r) = \textbf{false}$. Let $E' = [\![(\mathrm{rev}(\mathtt{S}), \varnothing, E)]\!]$ and $e' = [\![\mathsf{check}\, r\, \mathsf{then}\, e]\!] = \mathsf{check}\, r\, \mathsf{then}\, [\![e]\!]$, so that $(\mathtt{S}, e_e) \lhd E'[e']$. Then by Lemma 4.5 $E' \not\vdash r$, so $E'[e']$ is also stuck. □

It is now possible to demonstrate the principal result of this section, that $\lambda^{\mathtt{S}}_{\text{sec}}$ may be simulated in λ_{sec}, in a straightforward manner.

Theorem 4.2 (Simulation of $\lambda^{\mathtt{S}}_{\text{sec}}$ in λ_{sec}) *If* e *evaluates to* v *then* $[\![e]\!]$ *evaluates to* $[\![v]\!]$. *If* e *goes wrong then* $[\![e]\!]$ *goes wrong. If* e *diverges then* $[\![e]\!]$ *diverges.*

Proof. Suppose $nil, e \to^\star nil, v$. We have that $(nil, e) \lhd [\![e]\!]$ and $(nil, v) \lhd [\![v]\!]$ by definition of $\lhd$. But $[\![e]\!] \to^\star [\![v]\!]$ by Lemma 4.9, and $[\![v]\!]$ is a value by Lemma 4.10.

Suppose nil, e diverges. We have that $(nil, e) \lhd [\![e]\!]$; suppose on the contrary that there exists v such that $[\![e]\!] \to^\star v$. But by Lemma 4.9 there must exist e' such that $nil, e \to^\star \mathtt{S}, e'$ and $(\mathtt{S}, e') \lhd v$, and by Lemma 4.10 we have that e' is either a closure or an expression $\lambda x.e''$, which is a contradiction, either outright or because $(\mathtt{S}, \lambda x.e'')$ evaluates.

Suppose $nil, e \to^\star \mathtt{S}, e'$ and $\mathtt{S}, e'$ is stuck. We have that $(nil, e) \lhd [\![e]\!]$ by definition. Let $(\mathtt{S}, e') \lhd e''$; then $[\![e]\!] \to^\star e''$ by Lemma 4.9, and e'' is stuck by Lemma 4.10. □

4.3 The λ_{sec}-to-pml_B transformation

Now, we move on to the translation of λ_{sec} into pml_B, defined in Fig. 4.4. The distinguished identifiers s and $\_$ are assumed not to appear in source expressions. Notice that s may appear free in translated expressions. Translating an (unsigned) expression requires specifying the current principal p.

$$
\begin{array}{rcl}
[\![x]\!]_p & = & x \\
[\![\mathsf{fix}\, z.\lambda x.f]\!]_p & = & \mathsf{fix}\, z.\lambda x.\lambda s.[\![f]\!] \\
[\![e_1\, e_2]\!]_p & = & [\![e_1]\!]_p\, [\![e_2]\!]_p\, s \\
[\![\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2]\!]_p & = & \mathsf{let}\, x = [\![e_1]\!]_p \,\mathsf{in}\, [\![e_2]\!]_p \\
[\![\mathsf{enable}\, r \,\mathsf{in}\, e]\!]_p & = & \mathsf{let}\, s = (\{r\} \cap p) \vee s \,\mathsf{in}\, [\![e]\!]_p \\
[\![\mathsf{check}\, r \,\mathsf{then}\, e]\!]_p & = & \mathsf{let}\, \_ = s \ni r \,\mathsf{in}\, [\![e]\!]_p \\
[\![\mathsf{test}\, r \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2]\!]_p & = & ?_s\, r\, (\lambda s.[\![e_1]\!]_p)\, (\lambda s.[\![e_2]\!]_p) \\
[\![f]\!]_p & = & [\![f]\!] \\
\\
[\![p.e]\!] & = & \mathsf{let}\, s = p \wedge s \,\mathsf{in}\, [\![e]\!]_p
\end{array}
$$

Figure 4.4: λ_{sec}-to-pml_B transformation

One will often wish to translate an expression under minimal hypotheses, i.e. under the principal *nobody* and a void security context. To do so, we define $(\!| e |\!) = [\![e]\!]_{nobody}[\varnothing/s]$. Notice that s does not appear free in $(\!| e |\!)$. If e is closed, then so is $(\!| e |\!)$.

The idea behind the translation is simple: the variable s is bound at all times to the set of currently enabled resources. Every function accepts s as an extra parameter, because it must execute within its caller's security context. As a result, every function call has s as its second parameter. The constructs $\mathsf{enable}\, r \,\mathsf{in}\, e$ and $p.e$ cause s to be locally bound to a new value, reflecting the new security context; more specifically, the former enables r, while the latter disables all privileges not available to p. The constructs $\mathsf{check}\, r \,\mathsf{then}\, e$ and $\mathsf{test}\, r \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2$ are implemented simply by looking up the current value of s. In the latter, s is re-bound, within each branch, to the *same* value. This may appear superfluous at first sight, but has an important impact on typing, because it allows s to be given a different (more precise) type within each branch.

This translation can be viewed as a generalization of Wallach's security-passing style transformation [45] to a higher-order setting. Whereas Wallach advocated this idea as an implementation technique, with efficiency in mind, we use it only as a vehicle in the proof of our type systems. Here, efficiency is not at stake: it is sufficient that the translation scheme be correct. The next section is devoted to proving this fact.

4.3.1 Properties

A basic property of the translation is that s never appears free in the translation of a value. Furthermore, the translation of a value does not depend on the current principal, so we write $[\![v]\!]$ instead of $[\![v]\!]_p$.

Since λ_{sec} has no state, we are concerned only with the state-free subset of pml_B in this Chapter, and so define the syntactic sugar $e \rightarrow e' \triangleq e, \varnothing \rightarrow e', \varnothing$ for pml_B reductions, for brevity. For the purposes of our proofs, we need to isolate a particular sub-class of target language reductions, which we wish to view as "administrative" (in a sense to be explained later). Let $\rightarrow_\sim$ be the subset of $\rightarrow^\star$ defined by:

$$\begin{array}{rcll} a & ::= & R \mid R \vee a \mid R \wedge a & \\ \mathsf{let}\, s = a\, \mathsf{in}\, e & \rightarrow_\sim & e[R/s] & \text{if } a \rightarrow^\star R \\ E[e] & \rightarrow_\sim & E[e'] & \text{if } e \rightarrow_\sim e' \end{array}$$

Our first lemma expresses the fact that the translation *implements* the forward stack inspection algorithm of Fig. 4.3. It states that if $p, R, E \vdash R'$, then evaluating $[\![E[e]]\!]_p$ in a context where s is bound to R leads to evaluating $[\![e]\!]_{p'}$, for some p', in a context where s is bound to R'. Furthermore, this is a purely administrative reduction sequence. That is, it only affects the security context, and does not reflect any computational steps apparent in the original program. The proof of the lemma presents no difficulty, because of the close similarity between the definitions of the translation function and of the stack inspection algorithm.

Lemma 4.11 *Assume $p, R, S \vdash R'$ and $S = |E|$. Then, there exist a (target) evaluation context E' and a principal p' such that, for every source expression e,*

$$[\![E[e]]\!]_p[R/s] \rightarrow_\sim^\star E'[[\![e]\!]_{p'}[R'/s]]$$

Proof. By induction over the structure of E. Let θ and θ' stand for the substitutions $[R/s]$ and $[R'/s]$, respectively.

Case $E = []$. Then, $S = \epsilon$ and $R = R'$. Thus, picking $E' = []$ and $p' = p$ trivially satisfies our requirement.

Case $E = E_1\, e_1$. Then,

$$[\![E[e]]\!]_p\theta = [\![E_1[e]]\!]_p\theta\; [\![e_1]\!]_p\theta\; R$$

Furthermore, the induction hypothesis, applied to E_1, yields E_1' and p' such that $[\![E_1[e]]\!]_p\theta \rightarrow_\sim^\star E_1'[[\![e]\!]_{p'}\theta']$. So, picking $E' = E_1'\; [\![e_1]\!]_p\theta\; R$ fits the bill.

Case $E = v\, E_1$. This case is similar to the previous one. Apply the induction hypothesis to obtain E'_1 and p'. Then, pick $E' = [\![v]\!]\ E'_1\ R$. (E' is indeed an evaluation context, because $[\![v]\!]$ is a value.)

Case $E = \mathsf{let}\, x = E_1\, \mathsf{in}\, e_1$. This case is also similar. Apply the induction hypothesis to obtain E'_1 and p'. Then, pick $E' = \mathsf{let}\, x = E'_1\ \mathsf{in}\, [\![e_1]\!]_p\theta$.

Case $E = \mathsf{enable}\, r\, \mathsf{in}\, E_1$. Then, $S = r.S_1$, where $S_1 = \lfloor E_1 \rfloor$. Thus, from $p, R, S \vdash R'$, we may deduce $p, R_1, S_1 \vdash R'$, where R_1 stands for $R \cup (\{r\} \cap p)$. Define $\theta_1 = [R_1/s]$. Then,

$$\begin{array}{rcl} [\![E[e]]\!]_p\theta & = & \mathsf{let}\, s = (\{r\} \cap p) \vee R\, \mathsf{in}\, [\![E_1[e]]\!]_p \\ & \rightarrow_{\sim} & [\![E_1[e]]\!]_p\theta_1 \end{array}$$

Applying the induction hypothesis to E_1 yields E'_1, p' such that $[\![E_1[e]]\!]_p\theta_1 \rightarrow^{\star}_{\sim} E'_1[[\![e]\!]_{p'}\theta']$. So, picking $E' = E'_1$ meets our goal.

Case $E = p_1.E_1$. Then, $S = p_1.S_1$, where $S_1 = \lfloor E_1 \rfloor$. Thus, from $p, R, S \vdash R'$, we may deduce $p_1, R_1, S_1 \vdash R'$, where R_1 stands for $p_1 \cap R$. Define $\theta_1 = [R_1/s]$. Then,

$$\begin{array}{rcl} [\![E[e]]\!]_p\theta & = & \mathsf{let}\, s = p_1 \wedge R\, \mathsf{in}\, [\![E_1[e]]\!]_{p_1} \\ & \rightarrow_{\sim} & [\![E_1[e]]\!]_{p_1}\theta_1 \end{array}$$

Applying the induction hypothesis to E_1 yields E'_1, p' such that $[\![E_1[e]]\!]_{p_1}\theta_1 \rightarrow^{\star}_{\sim} E'_1[[\![e]\!]_{p'}\theta']$. So, picking $E' = E'_1$ meets our goal. $\square$

We now come to our central lemma, stating that, if a source expression e leads, in one computation step, to a source expression e', then the translation of e reduces, modulo administrative reductions, to the translation of e'.

Lemma 4.12 *$e \rightarrow e'$ implies $(\!|\, e\, |\!) \rightarrow^{\star} \cdot \mathrel{{}^{\star}_{\sim}\!\!\leftarrow} (\!|\, e'\, |\!)$.*

Proof. Because $e \rightarrow e'$, e and e' must be of the form $E[e_0]$ and $E[e'_0]$, respectively. Let $S = \lfloor E \rfloor$. There exists a unique R such that *nobody*$, \varnothing, S \vdash R$. Clearly, for any resource r, $E \vdash r$ is equivalent to $r \in R$. Define $\theta = [R/s]$. According to Lemma 4.11, there exist an evaluation context E' and a principal p such that, for any source expression e,

$$(\!|\, E[e]\, |\!) \rightarrow^{\star}_{\sim} E'[[\![e]\!]_p\theta]$$

Assume, for the time being, that $[\![e_0]\!]_p\theta \rightarrow^\star [\![e'_0]\!]_p\theta$ holds. Then, we have

$$\begin{array}{rcl} (\!|\, e \,|\!) = (\!|\, E[e_0] \,|\!) & \rightarrow^\star_\sim & E'[[\![e_0]\!]_p\theta] \\ & \rightarrow^\star & E'[[\![e'_0]\!]_p\theta] \\ & {}^\star_\sim\!\leftarrow & (\!|\, E[e'_0] \,|\!) = (\!|\, e' \,|\!) \end{array}$$

which is the desired result. Hence, there only remains to prove $[\![e_0]\!]_p\theta \rightarrow^\star [\![e'_0]\!]_p\theta$, which we will now do, by cases on the form of e_0 and e'_0.

Case $e_0 = (\mathsf{fix}\, z.\lambda x.f)\, v$, $e'_0 = f[v/x][\mathsf{fix}\, z.\lambda x.f/z]$. Then,

$$\begin{array}{rcll} [\![e_0]\!]_p\theta & = & [\![(\mathsf{fix}\, z.\lambda x.f)\, v]\!]_p\theta & \\ & = & ([\![\mathsf{fix}\, z.\lambda x.f]\!]\, [\![v]\!]\, s)\theta & \\ & = & (\mathsf{fix}\, z.\lambda x.\lambda s.[\![f]\!])\, [\![v]\!]\, R & \text{because } s \text{ cannot appear free in values} \\ & \rightarrow^2 & [\![f]\!][[\![v]\!]/x][[\![\mathsf{fix}\, z.\lambda x.f]\!]/z]\theta & \\ & = & [\![f[v/x][\mathsf{fix}\, z.\lambda x.f/z]]\!]\theta & \text{by a straightforward auxiliary lemma} \\ & = & [\![e'_0]\!]_p\theta & \end{array}$$

The auxiliary lemma mentioned above takes advantage of the fact that the translation of a value $[\![v]\!]_p$ does not depend upon the parameter p. We omit its proof.

Case $e_0 = \mathsf{let}\, x = v \,\mathsf{in}\, e_1$, $e'_0 = e_1[v/x]$. Then,

$$\begin{array}{rcll} [\![e_0]\!]_p\theta & = & [\![\mathsf{let}\, x = v \,\mathsf{in}\, e_1]\!]_p\theta & \\ & = & \mathsf{let}\, x = [\![v]\!] \,\mathsf{in}\, [\![e_1]\!]_p\theta & \text{because } s \text{ is not free in } [\![v]\!] \\ & \rightarrow & [\![e_1]\!]_p\theta[[\![v]\!]/x] & \\ & = & [\![e_1]\!]_p[[\![v]\!]/x]\theta & \\ & = & [\![e_1[v/x]]\!]_p\theta & \text{by the same auxiliary lemma} \\ & = & [\![e'_0]\!]_p\theta & \end{array}$$

Case $e_0 = \mathsf{enable}\, r \,\mathsf{in}\, v$, $e'_0 = v$. Then,

$$\begin{array}{rcl} [\![e_0]\!]_p\theta = [\![\mathsf{enable}\, r \,\mathsf{in}\, v]\!]_p\theta & = & \mathsf{let}\, s = (\{r\} \cap p) \vee R \,\mathsf{in}\, [\![v]\!] \\ & \rightarrow^2 & [\![v]\!] = [\![e'_0]\!]_p\theta \end{array}$$

Again, we take advantage of the fact that s does not occur free in $[\![v]\!]$.

Case $e_0 = \mathsf{check}\, r \,\mathsf{then}\, e_1$, $e'_0 = e_1$. We must have $E \vdash r$, hence $r \in R$. Then,

$$\begin{array}{rcll} [\![e_0]\!]_p\theta = [\![\mathsf{check}\, r \,\mathsf{then}\, e_1]\!]_p\theta & = & \mathsf{let}\, \_ = R \ni r \,\mathsf{in}\, [\![e_1]\!]_p\theta & \\ & \rightarrow^2 & [\![e_1]\!]_p\theta & \text{because } r \in R \\ & = & [\![e'_0]\!]_p\theta & \end{array}$$

Case $e_0 = \textsf{test}\, r\, \textsf{then}\, e_1\, \textsf{else}\, e_2$. Then, e_0' equals e_i, where $i = 1$ if $E \vdash r$ (or, equivalently, if $r \in R$), and $i = 2$ otherwise. Thus, we have

$$\begin{array}{rcl} [\![e_0]\!]_p\theta = [\![\textsf{test}\, r\, \textsf{then}\, e_1\, \textsf{else}\, e_2]\!]_p\theta & = & ?_R\, r\, (\lambda s.[\![e_1]\!]_p)\, (\lambda s.[\![e_2]\!]_p) \\ & \rightarrow^3 & (\lambda s.[\![e_i]\!]_p)\, R \\ & \rightarrow & [\![e_i]\!]_p\theta = [\![e_0']\!]_p\theta \end{array}$$

Case $e_0 = p_1.v$, $e_0' = v$. Then,

$$\begin{array}{rcl} [\![e_0]\!]_p\theta = [\![p_1.v]\!]_p\theta & = & \textsf{let}\, s = p_1 \wedge R\, \textsf{in}\, [\![v]\!] \\ & \rightarrow^2 & [\![v]\!] = [\![e_0']\!]_p\theta \end{array}$$

Again, we take advantage of the fact that s does not occur free in $[\![v]\!]_p$, and of the fact that this expression does not depend on p. □

This result is easily generalized to reduction sequences of arbitrary length:

Lemma 4.13 $e \rightarrow^\star e'$ *implies* $(\!| e |\!) \rightarrow^\star \cdot \overset{\star}{\sim}\!\leftarrow (\!| e' |\!)$.

Proof. By induction on the length of the reduction sequence $e \rightarrow^\star e'$. In the base case, we have $e = e'$, and the result is immediate. In the inductive case, we have $e \rightarrow e_1 \rightarrow^\star e'$. By applying Lemma 4.12, on the one hand, and the induction hypothesis, on the other hand, we obtain

$$(\!| e |\!) \rightarrow^\star \cdot \overset{\star}{\sim}\!\leftarrow (\!| e_1 |\!) \rightarrow^\star \cdot \overset{\star}{\sim}\!\leftarrow (\!| e' |\!)$$

Because the operational semantics of the target language is deterministic, one of the two reduction sequences starting at $(\!| e_1 |\!)$ above must be a sub-sequence of the other. In either case, the diagram collapses down to

$$(\!| e |\!) \rightarrow^\star \cdot \overset{\star}{\sim}\!\leftarrow (\!| e' |\!)$$

hence the result. □

As a corollary, we obtain a soundness theorem for the translation. It essentially states that security-passing style is a valid implementation of the Java stack inspection discipline.

Theorem 4.3 (λ_{sec}-to-pml_B transformation correctness) *If $e \rightarrow^\star v$, then $(\!| e |\!) \rightarrow^\star (\!| v |\!)$. If e goes wrong, then $(\!| e |\!)$ goes wrong. If e diverges, then $(\!| e |\!)$ diverges.*

Proof. First, assume e reduces to a value v. Then, Lemma 4.13 yields $(\!| e |\!) \rightarrow^\star \cdot \overset{\star}{\sim}\!\leftarrow (\!| v |\!)$. Because $(\!| v |\!)$ is a value, this diagram collapses down to $(\!| e |\!) \rightarrow^\star (\!| v |\!)$.

Second, assume e goes wrong. Then, $e \rightarrow^\star e'$, where e' is stuck, holds. We prove that $(\!| e |\!)$ goes wrong by induction on the length of this reduction sequence.

In the base case, we have $e = e'$, i.e. e is stuck. Therefore, we have that e must be of the form $E[\mathsf{check}\, r\, \mathsf{then}\, e_1]$, where $\neg(E \vdash r)$. Let $S = |E|$. There exists a unique R' such that $\textit{nobody}, \varnothing, S \vdash R'$. Necessarily, $r \notin R'$. According to Lemma 4.11, $(\!| e |\!)$ may be reduced to a term of the form $E'[[\![\mathsf{check}\, r\, \mathsf{then}\, e_1]\!]_{p'}\theta']$, where $\theta' = [R'/s]$. It is easy to check that such a term is stuck. Hence, $(\!| e |\!)$ goes wrong.

In the inductive case, we have $e \rightarrow e_1 \rightarrow^\star e'$. Our induction hypothesis shows that $(\!| e_1 |\!)$ goes wrong. Furthermore, Lemma 4.12 shows that $(\!| e |\!)$ reduces to some reduct of $(\!| e_1 |\!)$. Because reduction is deterministic, $(\!| e |\!)$ must go wrong as well. The result follows.

Third, assume e admits an infinite reduction sequence. This sequence must involve an infinite number of β-reduction steps, because the semantics of λ_{sec}, deprived of the β-reduction rule, is terminating. Now, a slight generalization of Lemma 4.13 states that if a reduction from e to e' involves k β-reduction steps, then $(\!| e |\!)$ reduces to $(\!| e' |\!)$ (modulo administrative reductions) in at least k β-reduction steps. (The proof, which we omit, hinges on the fact that β-reduction is not an administrative reduction.) This implies that $(\!| e |\!)$ admits an infinite reduction sequence as well. $\square$

4.4 Types for λ_{sec}

In this section we introduce a let-polymorphic type analysis for λ_{sec}. In fact, using the type machinery in place for pml_B and Theorem 4.3, we are able to define more than one type system for the language. We have argued that the transformational approach allows us to easily develop a type system for λ_{sec}; we will demonstrate this in Sect. 4.4.1 by immediately obtaining an *indirect* type analysis for λ_{sec}, via composition of the λ_{sec}-to-pml_B transformation and pml_B type judgements. But as even more significant evidence of the benefits of the transformational approach, it is made apparent in Sect. 4.4.2 that proof of correctness for a *direct* λ_{sec} type analysis is significantly easier using this approach.

4.4.1 Indirect types

The type systems $\mathcal{S}_i^{\mathit{rel}}$ for pml_B were specified in Definition 3.2. Sect. 4.3 defined a translation of λ_{sec} into pml_B. Composing the two automatically gives rise to a type system for λ_{sec}, also called $\mathcal{S}_i^{\mathit{rel}}$ for simplicity, whose safety is a direct consequence of Theorems 4.3 and 3.2.

Definition 4.4 *Let e be a closed λ_{sec} expression. By definition, $C, \Gamma \vdash e : \sigma$ holds if and only if $C, \Gamma \vdash (\!| e |\!) : \sigma$ holds.*

Theorem 4.4 *If $C, \Gamma \vdash e : \sigma$ holds, then e does not go wrong.*

Turning type safety into a trivial corollary was the principal motivation for basing our approach on a transformation. Indeed, because Theorem 4.3 concerns untyped terms, its proof is straightforward, and constitutes the principal proof effort for this λ_{sec} soundness result. It was necessary to prove δ-typability for pml_B in Lemma 3.5, which did involve types, but this is a subjective requirement of the target language. And, as we will show in Chapter 4, the pml_B language, along with its type soundness, can be re-used as the target language in other applications of the transformational approach.

4.4.2 Direct types

Definition 4.4, although simple, is not a *direct* definition of typing for λ_{sec}— where by direct we mean a type analysis of λ_{sec} expressions themselves, rather than their transformed images. There are several reasons why a direct analysis is desirable, chief among them that the λ_{sec}-to-pml_B transformation might be inefficient for an implementation of λ_{sec}, and also type error reporting would be problematic. We thus will give rules which allow typing λ_{sec} expressions without explicitly translating them into pml_B. These direct rules can be *derived* in a rather systematic way from the definition of $\mathcal{S}_i^{rel}$ and the definition of the translation. This method will allow us to prove type safety for the direct λ_{sec} type analysis by a proof of correspondence between the direct and indirect analyses. The proof is straightforward, far simpler than a direct proof of type soundness for λ_{sec} would be.

For the direct system we re-use the type and constraint language of RS defined in Fig. 3.3, and the abbreviated set types of Sect. 3.2.4. For clarity in the subsequent presentation, we constrain τ to range over types of kind *Type*, φ to range over types of kind *Cap*, and ρ to range over abbreviated row types of kind *Row*$(c)_\star$; the $\star$ symbol indicates an arbitrary principal. Further, we constrain ς to range over abbreviated set types of the form $\{\rho\}$; in the direct type system, ς is used to represent some security context, i.e. a set of available resources. To recover the more intuitive and readable notation proposed in Sect. 1.3, we define the macro $\tau_1 \xrightarrow{\varsigma_2} \tau_2 \triangleq \tau_1 \to \varsigma_2 \to \tau_2$; as an artifact of the λ_{sec}-to-pml_B translation, where all functions are given an additional parameter, and the fact that we derive direct types from the indirect, all function types in the direct system are of this form.

$$\textsc{Var}\quad \frac{\Gamma(x) = \sigma}{p, \varsigma, \Gamma \vdash x : \sigma}$$

$$\textsc{Abs}\quad \frac{\star, \varsigma_2, (\Gamma; z : \tau_1 \xrightarrow{\varsigma_2} \tau_2; x : \tau_1) \vdash f : \tau_2}{p, \varsigma_1, \Gamma \vdash \mathsf{fix}\, z.\lambda x.f : \tau_1 \xrightarrow{\varsigma_2} \tau_2}$$

$$\textsc{App}\quad \frac{p, \varsigma, \Gamma \vdash e_1 : \tau_2 \xrightarrow{\varsigma} \tau \qquad p, \varsigma, \Gamma \vdash e_2 : \tau_2}{p, \varsigma, \Gamma \vdash e_1\, e_2 : \tau}$$

$$\textsc{Let}\quad \frac{p, \varsigma, \Gamma \vdash e_1 : \sigma \qquad p, \varsigma, (\Gamma; x : \sigma) \vdash e_2 : \tau}{p, \varsigma, \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau}$$

$$\forall\,\textsc{Intro}\quad \frac{p, \varsigma, \Gamma \vdash e : \tau \qquad \bar{\alpha} \cap \mathsf{fv}(\varsigma, \Gamma) = \varnothing}{p, \varsigma, \Gamma \vdash e : \forall \bar{\alpha}.\tau}$$

$$\forall\,\textsc{Elim}\quad \frac{p, \varsigma, \Gamma \vdash e : \forall \bar{\alpha}.\tau}{p, \varsigma, \Gamma \vdash e : \tau[\bar{\tau}/\bar{\alpha}]}$$

$$\textsc{Enable Failure}\quad \frac{p, \{\rho\}, \Gamma \vdash e : \tau \qquad r \notin p}{p, \{\rho\}, \Gamma \vdash \mathsf{enable}\, r\, \mathsf{in}\, e : \tau}$$

$$\textsc{Enable Success}\quad \frac{p, \{r+, \rho\}, \Gamma \vdash e : \tau \qquad r \in p}{p, \{r\varphi, \rho\}, \Gamma \vdash \mathsf{enable}\, r\, \mathsf{in}\, e : \tau}$$

$$\textsc{Check}\quad \frac{p, \{r+, \rho\}, \Gamma \vdash e : \tau}{p, \{r+, \rho\}, \Gamma \vdash \mathsf{check}\, r\, \mathsf{then}\, e : \tau}$$

$$\textsc{Test}\quad \frac{p, \{r+, \rho\}, \Gamma \vdash e_1 : \tau \qquad p, \{r-, \rho\}, \Gamma \vdash e_2 : \tau}{p, \{r\varphi, \rho\}, \Gamma \vdash \mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : \tau}$$

$$\textsc{Sign}\quad \frac{p, \{r_1\varphi_1, \ldots, r_n\varphi_n, \varnothing\}, \Gamma \vdash e : \tau \qquad p = \{r_1, \ldots, r_n\}}{\star, \{r_1\varphi_1, \ldots, r_n\varphi_n, \rho\}, \Gamma \vdash p.e : \tau}$$

Figure 4.5: Typing rules for λ_{sec} derived from $\mathcal{S}_1^{=}$

$$\textsc{Var}\quad \frac{\Gamma(x)=\sigma \qquad C \Vdash \sigma}{p,\varsigma,C,\Gamma \vdash x : \sigma}$$

$$\textsc{Sub}\quad \frac{p,\varsigma,C,\Gamma \vdash e : \tau \qquad C \Vdash \tau \leq \tau'}{p,\varsigma,C,\Gamma \vdash e : \tau'}$$

$$\textsc{Abs}\quad \frac{\star,\varsigma_2,C,(\Gamma; x : \tau_1) \vdash f : \tau_2}{p,\varsigma_1,C,\Gamma \vdash \lambda x.f : \tau_1 \rightarrow \varsigma_2 \rightarrow \tau_2}$$

$$\textsc{App}\quad \frac{p,\varsigma,C,\Gamma \vdash e_1 : \tau_2 \rightarrow \varsigma \rightarrow \tau \qquad p,\varsigma,C,\Gamma \vdash e_2 : \tau_2}{p,\varsigma,C,\Gamma \vdash e_1\, e_2 : \tau}$$

$$\textsc{Let}\quad \frac{p,\varsigma,C,\Gamma \vdash e_1 : \sigma \qquad p,\varsigma,C,(\Gamma; x : \sigma) \vdash e_2 : \tau}{p,\varsigma,C,\Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \tau}$$

$$\forall\ \textsc{Intro}\quad \frac{p,\varsigma,C \wedge D,\Gamma \vdash e : \tau \qquad \bar{\alpha} \cap \mathsf{fv}(\varsigma,C,\Gamma) = \varnothing}{p,\varsigma,C \wedge \exists\bar{\alpha}.D,\Gamma \vdash e : \forall\bar{\alpha}[D].\tau}$$

$$\forall\ \textsc{Elim}\quad \frac{p,\varsigma,C,\Gamma \vdash e : \forall\bar{\alpha}[D].\tau}{p,\varsigma,C \wedge D,\Gamma \vdash e : \tau}$$

$$\exists\ \textsc{Intro}\quad \frac{p,\varsigma,C,\Gamma \vdash e : \sigma \qquad \bar{\alpha} \cap \mathsf{fv}(\varsigma,\Gamma,\sigma) = \varnothing}{p,\varsigma,\exists\bar{\alpha}.C,\Gamma \vdash e : \sigma}$$

$$\textsc{Enable Failure}\quad \frac{p,\{\rho\},C,\Gamma \vdash e : \tau \qquad r \notin p}{p,\{\rho\},C,\Gamma \vdash \mathsf{enable}\, r \,\mathsf{in}\, e : \tau}$$

$$\textsc{Enable Success}\quad \frac{p,\{r+,\rho\},C,\Gamma \vdash e : \tau \qquad r \in p}{p,\{r\varphi,\rho\},C,\Gamma \vdash \mathsf{enable}\, r \,\mathsf{in}\, e : \tau}$$

$$\textsc{Check}\quad \frac{p,\{r+,\rho\},C,\Gamma \vdash e : \tau}{p,\{r+,\rho\},C,\Gamma \vdash \mathsf{check}\, r \,\mathsf{then}\, e : \tau}$$

$$\textsc{Test}\quad \frac{\begin{array}{ccc} p,\{r+,\rho_1\},C,\Gamma \vdash e_1 : \tau_1 & p,\{r-,\rho_2\},C,\Gamma \vdash e_2 : \tau_2 & C \Vdash \text{if } + \leq \varphi \text{ then } \rho \leq \rho_1 \\ C \Vdash \text{if } - \leq \varphi \text{ then } \rho \leq \rho_2 & C \Vdash \text{if } + \leq \varphi \text{ then } \tau_1 \leq \tau & C \Vdash \text{if } - \leq \varphi \text{ then } \tau_2 \leq \tau \end{array}}{p,\{r\varphi,\rho\},C,\Gamma \vdash \mathsf{test}\, r \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2 : \tau}$$

$$\textsc{Sign}\quad \frac{p,\{r_1\varphi_1,\ldots,r_n\varphi_n,\varnothing\},C,\Gamma \vdash e : \tau \qquad p = \{r_1,\ldots,r_n\}}{\star,\{r_1\varphi_1,\ldots,r_n\varphi_n,\rho\},C,\Gamma \vdash p.e : \tau}$$

Figure 4.6: Typing rules for λ_{sec} derived from $\mathcal{S}_2^{\leq}$

Figure 4.6 gives rules for the system derived from $\mathcal{S}_2^{\leq}$, the most complex element in our array of type systems. Judgements have the form $p, \varsigma, C, \Gamma \vdash e : \sigma$.

Fig. 4.5 gives derived rules for $\mathcal{S}_1^{=}$, the simplest of our type systems. There, all constraints are equations. As a result, all type information can be represented in term form, rather than in constraint form [41]. We exploit this fact to give a simple presentation of the derived rules. Type schemes have the form $\forall\bar{\alpha}.\tau$, and judgements have the form $p, \varsigma, \Gamma \vdash e : \sigma$.

4.4.3 Direct type safety and optimizations

We will prove progress and type safety for $\mathcal{S}_1^{=}$ in this section, Theorems 4.6 and 4.7. The proofs are quite straightforward, requiring only that we prove a correspondence between the direct and indirect systems, Theorem 4.5, rather than prove non-trivial subject reduction result from scratch. We observe that the same result follows for $\mathcal{S}_2^{\leq}$ in a similar manner. Again, the ease of these results demonstrates the effectiveness of the transformational approach. We also discuss some run-time optimizations that can be performed as a consequence of these results.

First, we may prove the correspondence result in one direction, by showing that if a type judgement is derivable in the direct type system, then the same type judgement is derivable in the indirect one.

Lemma 4.14 *If $p, \varsigma, \Gamma \vdash e : \sigma$ is derivable then so is* **true**, $(\Gamma; s : \varsigma) \vdash [\![e]\!]_p : \sigma$.

Proof. Let $\Gamma' = (\Gamma; s : \varsigma)$; then the proof proceeds by induction on the height of the derivation of $p, \varsigma, \Gamma \vdash e : \tau$ and case analysis on the final step of the judgement:

Case VAR. In this case e is a variable x and $\Gamma(x) = \sigma$ by VAR, so $\Gamma'(x) = \sigma$ since $x \neq s$ by definition by definition of the translation. Therefore the Lemma holds in this case by the HM(X) VAR rule, since $[\![x]\!]_p = x$.

Case ABS. In this case $e = \mathsf{fix}\, z.\lambda x.f$, $\sigma = \tau_1 \xrightarrow{\varsigma'} \tau_2$ and:

$$\star, \varsigma', (\Gamma; z : \tau_1 \xrightarrow{\varsigma'} \tau_2; x : \tau_1) \vdash f : \tau_2$$

is derivable. But since:

$$(\Gamma; z : \tau_1 \xrightarrow{\varsigma'} \tau_2; x : \tau_1; s : \varsigma') = (\Gamma'; z : \tau_1 \xrightarrow{\varsigma'} \tau_2; x : \tau_1; s : \varsigma')$$

therefore by the induction hypothesis the judgement:

$$\textbf{true}, (\Gamma'; z : \tau_1 \xrightarrow{\varsigma'} \tau_2; x : \tau_1; s : \varsigma') \vdash [\![f]\!]_p : \tau_2$$

is derivable, thus $\mathbf{true}, \Gamma' \vdash \mathsf{fix}\, z.\lambda x.\lambda s.[\![f]\!]_p : \tau_1 \to \varsigma' \to \tau_2$ is derivable by two applications of the HM(X) ABS rule, so the Lemma holds by definition of $[\![e]\!]_p$ in this case.

Case APP. In this case $e = e_1 e_2$, $\sigma = \tau$ and $p, \varsigma, \Gamma \vdash e_1 : \tau_2 \xrightarrow{\varsigma} \tau$ and $p, \varsigma, \Gamma \vdash e_2 : \tau_2$ are derivable. But then by the induction hypothesis it is the case that $\mathbf{true}, \Gamma' \vdash [\![e_1]\!]_p : \tau_2 \to \varsigma \to \tau$ and $\mathbf{true}, \Gamma' \vdash [\![e_2]\!]_p : \tau_2$ are derivable, so that $\mathbf{true}, \Gamma' \vdash [\![e_1]\!]_p [\![e_2]\!]_p : \varsigma \to \tau_2$ is derivable by the HM(X) APP rule. But $\mathbf{true}, \Gamma' \vdash s : \varsigma$ by the HM(X) VAR rule, hence $\mathbf{true}, \Gamma' \vdash [\![e_1]\!]_p [\![e_2]\!]_p\, s : \tau_2$ by the HM(X) APP rule, so the Lemma holds by definition of $[\![e]\!]_p$ in this case.

Case LET. In this case $e = \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$, and $p, \varsigma, \Gamma \vdash e_1 : \sigma'$ and $p, \varsigma, (\Gamma; x : \sigma') \vdash e_2 : \sigma$ are derivable. But then $\mathbf{true}, \Gamma' \vdash [\![e_1]\!]_p : \sigma'$ and $\mathbf{true}, (\Gamma'; x : \sigma') \vdash [\![e_2]\!]_p : \sigma$ are derivable by the induction hypothesis, so the Lemma holds by the HM(X) LET rule and the definition of $[\![e]\!]_p$ in this case.

Case $\forall$ INTRO. In this case $\sigma = \forall \bar{\alpha}[\mathbf{true}].\tau$ where $\bar{\alpha} \cap \mathrm{fv}(\varsigma, \Gamma) = \varnothing$ and $p, \varsigma, \Gamma \vdash e : \tau$ is derivable. But then $\bar{\alpha} \cap \mathrm{fv}(\mathbf{true}, \Gamma') = \varnothing$. Now, by the induction hypothesis, $\mathbf{true}, \Gamma' \vdash [\![e]\!]_p : \tau$ is derivable, so also $\mathbf{true} \wedge \mathbf{true}, \Gamma' \vdash [\![e]\!]_p : \tau$ since $\mathbf{true} \wedge \mathbf{true}$ is equivalent to $\mathbf{true}$. Thus by the HM(X) $\forall$ INTRO rule it is the case that $\mathbf{true} \wedge \exists \bar{\alpha}.\mathbf{true}, \Gamma' \vdash [\![e]\!]_p : \sigma$ is derivable, so the Lemma holds in this case since $\mathbf{true} \wedge \exists \bar{\alpha}.\mathbf{true}$ is equivalent to $\mathbf{true}$.

Case $\forall$ ELIM. In this case $\sigma = [\bar{\tau}/\bar{\alpha}]\tau$ and $p, \varsigma, \Gamma \vdash e : \forall \bar{\alpha}[\mathbf{true}].\tau$ is derivable. But then $\mathbf{true}, \Gamma' \vdash [\![e]\!]_p : \forall \bar{\alpha}[\mathbf{true}].\tau$ is derivable by the induction hypothesis, and $\mathbf{true} \Vdash [\bar{\tau}/\bar{\alpha}]\mathbf{true}$ since $[\bar{\tau}/\bar{\alpha}]\mathbf{true} = \mathbf{true}$, so this case holds by the HM(X) $\forall$ ELIM rule.

Case ENABLE FAILURE. In this case $e = \mathsf{enable}\, r \,\mathsf{in}\, e'$ where $r \notin p$ so that $p \cap \{r\} = \varnothing$, $\sigma = \tau$ and $p, \varsigma, \Gamma \vdash e' : \tau$ is derivable. Now, by Lemma 3.6 and $\forall$ ELIM it is the case that $\mathbf{true}, \Gamma' \vdash \vee\varnothing : \varsigma \to \varsigma$ is derivable, and $\mathbf{true}, \Gamma' \vdash s : \varsigma$ is derivable by the HM(X) VAR rule, therefore $\mathbf{true}, \Gamma' \vdash \varnothing \vee s : \varsigma$ by the HM(X) APP rule. But $\mathbf{true}, \Gamma' \vdash [\![e']\!]_p : \tau$ is derivable by the induction hypothesis, so also $\mathbf{true}, (\Gamma'; s : \varsigma) \vdash [\![e']\!]_p : \sigma$ since $\Gamma' = (\Gamma'; s : \varsigma)$, therefore the Lemma holds by the HM(X) LET rule and the definition of $[\![e]\!]_p$ in this case.

Case ENABLE SUCCESS. In this case $e = \mathsf{enable}\, r \,\mathsf{in}\, e'$ where $r \in p$ so that $p \cap \{r\} = \{r\}$, $\varsigma = \{r\varphi, \rho\}$, $\sigma = \tau$ and $p, \{r{+}, \rho\}, \Gamma \vdash e' : \tau$ is derivable. Now, by Lemma 3.6 and $\forall$ ELIM it is the case that $\mathbf{true}, \Gamma' \vdash \vee\{r\} : \{r\varphi, \rho\} \to \{r{+}, \rho\}$ is derivable, and $\mathbf{true}, \Gamma' \vdash s : \varsigma$ is derivable by the HM(X) VAR rule, therefore $\mathbf{true}, \Gamma' \vdash \{r\} \vee s : \{r{+}, \rho\}$ by the HM(X) APP rule. But $\mathbf{true}, (\Gamma; s : \{r{+}, \rho\}) \vdash [\![e']\!]_p : \tau$ is derivable by the induction hypothesis, so also $\mathbf{true}, (\Gamma'; s : \{r{+}, \rho\}) \vdash [\![e']\!]_p : \sigma$ since $(\Gamma; s : \{r{+}, \rho\}) = (\Gamma'; s : \{r{+}, \rho\})$, therefore the Lemma holds by the HM(X) LET rule and the definition of $[\![e]\!]_p$ in this case.

Case CHECK. In this case $e = \mathsf{enable}\, r \,\mathsf{in}\, e'$ and $\sigma = \tau$, $\varsigma = \{r{+}, \rho\}$ and $p, \varsigma, \Gamma \vdash e' : \tau$

is derivable. But by the induction hypothesis it is the case that **true**, $\Gamma' \vdash [\![e']\!]_p : \tau$ is derivable, so also **true**, $(\Gamma'; \_ : \sigma') \vdash [\![e']\!]_p : \tau$ for any σ', since $\_$ does not appear in e', therefore the Lemma holds by the HM(X) LET rule and the definition of $[\![e]\!]_p$ in this case.

Case TEST. In this case $e = \mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ and $\sigma = \tau$, $\varsigma = \{r\varphi, \rho\}$ and $p, \{r+, \rho\}, \Gamma \vdash e_1 : \tau$ and $p, \{r-, \rho\}, \Gamma \vdash e_2 : \tau$ are derivable. Let $\Gamma'' = (\Gamma)$; then by the induction hypothesis it is the case that **true**, $(\Gamma''; s : \{r+, \rho\}) \vdash [\![e_1]\!]_p : \tau$ and **true**, $(\Gamma''; s : \{r-, \rho\}) \vdash [\![e_2]\!]_p : \tau$, so also **true**, $(\Gamma'; s : \{r+, \rho\}) \vdash [\![e_1]\!]_p : \tau$ and **true**, $(\Gamma'; s : \{r-, \rho\}) \vdash [\![e_2]\!]_p : \tau$, since $(\Gamma''; s : \{r\varphi, \rho\}) = (\Gamma'; s : \{r\varphi, \rho\})$. Thus by the HM($X$) ABS rule it is the case that **true**, $\Gamma' \vdash \lambda s.[\![e_1]\!]_p : \{r+, \rho\} \to \tau$ and **true**, $\Gamma' \vdash \lambda s.[\![e_2]\!]_p : \{r-, \rho\} \to \tau$ are derivable. Now, by the definition of Δ_1, and CONST and $\forall$ ELIM it is the case that **true**, $\Gamma' \vdash ?_r : \{r\varphi, \rho\} \to (\{r+, \rho\} \to \tau) \to (\{r-, \rho\} \to \tau) \to \tau$ is derivable, and **true**, $\Gamma' \vdash s : \{r\varphi, \rho\}$ is derivable by the HM(X) VAR rule, therefore the Lemma holds by three applications of the HM(X) APP rule and the definition of $[\![e]\!]_p$ in this case.

Case SIGN. In this case $p = \star$, $e = p'.e'$, $\varsigma = \{r_1\varphi_1, \ldots, r_n\varphi_n, \rho\}$, $\sigma = \tau$ and $p', \{r_1\varphi_1, \ldots, r_n\varphi_n, \varnothing\}, \Gamma \vdash e' : \tau$, where $p' = \{r_1, \ldots, r_n\}$. Let $\varsigma' = \{r_1\varphi_1, \ldots, r_n\varphi_n, \varnothing\}$; then by the induction hypothesis it is the case that **true**, $(\Gamma; s : \varsigma') \vdash [\![e']\!]_{p'} : \tau$ is derivable, so also **true**, $(\Gamma'; s : \varsigma') \vdash [\![e']\!]_{p'} : \tau$, since $(\Gamma; s : \varsigma') = (\Gamma'; s : \varsigma')$. Now by Lemma 3.6 and $\forall$ ELIM it is the case that **true**, $\Gamma' \vdash \wedge p' : \varsigma \to \varsigma'$ is derivable, and **true**, $\Gamma' \vdash s : \varsigma$ is derivable by VAR, so **true**, $\Gamma' \vdash p' \wedge s : \varsigma'$ is derivable by the HM(X) APP rule, therefore the Lemma holds by the HM(X) LET rule and the definition of $[\![e]\!]_p$ in this case. □

Now, we prove the other direction, that if a type judgement is derivable in the indirect λ_{sec} type system, then it is derivable in the direct system, Lemma 4.18. First, a couple more utility Lemmas are stated:

Lemma 4.15 *A type scheme* $\forall\bar{\alpha}[D].\tau$ *is* dead *iff* $\bar{\alpha} \cap \mathrm{fv}(D, \tau) = \varnothing$. *If* $C, (\Gamma; x : \forall\bar{\alpha}[D].\tau) \vdash e : \sigma$ *is derivable and* $\forall\bar{\alpha}[D].\tau$ *is dead, then* $C, (\Gamma; x : \tau) \vdash e : \sigma$ *is also derivable in the same height.*

Lemma 4.16 *If* $C, (\Gamma; x : \sigma) \vdash e : \sigma'$ *is derivable and* x *does not occur free in e, then* $C, \Gamma \vdash e : \sigma'$ *is also derivable in the same height.*

Lemma 4.17 *If* $C, (\Gamma) \vdash [\![\mathsf{fix}\, z.\lambda x.f]\!]_p : \tau$ *is derivable, then* τ *is of the form* $\tau_1 \to \varsigma \to \tau_2$.

Proof. This property holds by virtue of the transformation, since f must be of the form $p'.e$, hence:

$$[\![\mathsf{fix}\, z.\lambda x.f]\!]_p = \mathsf{fix}\, z.\lambda x.\lambda s.\mathsf{let}\, s = p' \wedge s\, \mathsf{in}\, [\![e]\!]_{p'}$$

This form ensures that all function types reflect the two arguments of transformed terms, with the type of the second argument s being a set type ς, since s is always the argument of a set intersection in the transformation. $\square$

Thus, we may prove the correspondence in the other direction as follows. In this Lemma, we abbreviate type schemes $\forall\bar{\alpha}[\textbf{true}].\tau$ as $\forall\bar{\alpha}.\tau$ and judgements $\textbf{true}, \Gamma \vdash e : \sigma$ as $\Gamma \vdash e : \sigma$, omitting the trivial requirement **true** $\Vdash$ **true** from instances of $\forall$ ELIM and VAR:

Lemma 4.18 *If* $(\Gamma; s : \varsigma) \vdash [\![e]\!]_p : \tau$ *is derivable in HM*$(\mathrm{RS}^=)$*, then* $p, \varsigma, \Gamma \vdash e : \tau$ *is derivable in* $\mathcal{S}_1^=$.

Proof. By Lemma 2.6 and definition of $\mathrm{RS}^=$, it is the case that $(\Gamma; s : \varsigma) \vdash [\![e]\!]_p : \tau$ follows by a syntax-directed rule and at most one instance of $\forall$ ELIM. Let $\Gamma' = (\Gamma; s : \varsigma)$; the proof then proceeds by induction on the height of the derivation of $\Gamma' \vdash [\![e]\!]_p : \tau$ and case analysis on $[\![e]\!]_p$:

Case $[\![e]\!]_p = x$. In this case $e = x \neq s$ and $x \in \mathrm{dom}(\Gamma)$ by definition of the translation. By Lemma 2.6 we have a derivation of the following form, where $\tau = [\bar{\tau}/\bar{\alpha}]\tau'$:

$$\dfrac{\dfrac{\Gamma'(x) = \forall\bar{\alpha}.\tau'}{\Gamma' \vdash x : \forall\bar{\alpha}.\tau'}}{\Gamma' \vdash x : [\bar{\tau}/\bar{\alpha}]\tau'}$$

But then by definition of $\mathcal{S}_1^=$ we have the following derivation:

$$\dfrac{\dfrac{\Gamma(x) = \forall\bar{\alpha}.\tau'}{p, \varsigma, \Gamma \vdash x : \forall\bar{\alpha}.\tau'}}{p, \varsigma, \Gamma \vdash x : [\bar{\tau}/\bar{\alpha}]\tau'}$$

Therefore, this case holds.

Case $[\![e]\!]_p = \mathsf{fix}\, z.\lambda x.\lambda s.[\![f]\!]_p$. In this case $e = \mathsf{fix}\, z.\lambda x.f$, and by Lemma 2.6 we have the following subderivations, where τ is of the form $\tau_1 \to \varsigma' \to \tau_2$ by Lemma 4.17:

$$\dfrac{\dfrac{\Gamma'; z : \tau_1 \to \varsigma' \to \tau_2; x : \tau_1; s : \varsigma' \vdash [\![f]\!]_p : \tau_2}{\Gamma'; z : \tau_1 \to \varsigma' \to \tau_2; x : \tau_1 \vdash \lambda s.[\![f]\!]_p : \varsigma' \to \tau_2}}{\Gamma' \vdash \mathsf{fix}\, z.\lambda x.\lambda s.[\![f]\!]_p : \tau_1 \to \varsigma' \to \tau_2}$$

But since:

$$(\Gamma'; z : \tau_1 \to \varsigma' \to \tau_2; x : \tau_1; s : \varsigma') = (\Gamma; z : \tau_1 \to \varsigma' \to \tau_2; x : \tau_1; s : \varsigma')$$

therefore by the induction hypothesis we have that $p, \varsigma', (\Gamma; z : \tau_1 \to \varsigma' \to \tau_2; x : \tau_1) \vdash f : \tau_2$ is derivable, so that $p, \varsigma, \Gamma \vdash \mathsf{fix}\, z.\lambda x.f : \tau_1 \to \varsigma' \to \tau_2$ is derivable by ABS.

Case $[\![e]\!]_p = [\![e_1]\!]_p[\![e_2]\!]_p s$. In this case $e = e_1 e_2$, and by Lemma 2.6 we have a subderivation of the following form:

$$\dfrac{\dfrac{\Gamma' \vdash [\![e_1]\!]_p : \tau' \to \varsigma \to \tau \qquad \Gamma' \vdash [\![e_2]\!]_p : \tau'}{\Gamma' \vdash [\![e_1]\!]_p[\![e_2]\!]_p : \varsigma \to \tau} \qquad \dfrac{\Gamma'(s) = \varsigma}{\Gamma' \vdash s : \varsigma}}{\Gamma' \vdash [\![e_1]\!]_p[\![e_2]\!]_p s : \tau}$$

But then by the induction hypothesis we have that $p, \varsigma, \Gamma \vdash e_1 : \tau' \to \varsigma \to \tau$ and $p, \varsigma, \Gamma \vdash e_2 : \tau'$ are derivable in $\mathcal{S}_1^=$, hence $p, \varsigma, \Gamma \vdash e_1 e_2 : \tau$ is derivable in $\mathcal{S}_1^=$ by APP.

Case $[\![e]\!]_p = \mathsf{let}\, x = [\![e_1]\!]_p \,\mathsf{in}\, [\![e_2]\!]_p$. In this case $e = \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$, and we have a subderivation of the following form:

$$\dfrac{\Gamma' \vdash [\![e_1]\!]_{:}\sigma \qquad (\Gamma; s : \varsigma; x : \sigma) \vdash [\![e_2]\!]_{:}\tau}{\Gamma' \vdash \mathsf{let}\, x = [\![e_1]\!]_p \,\mathsf{in}\, [\![e_2]\!]_p : \tau}$$

But since $x \neq s$ we have $(\Gamma; s : \varsigma; x : \sigma) = (\Gamma; x : \sigma; s : \varsigma)$, therefore by the induction hypothesis we have that both $p, \varsigma, \Gamma \vdash e_1 : \sigma$ and $p, \varsigma, (\Gamma; x : \sigma) \vdash e_2 : \tau$ are derivable, so this case follows by LET in $\mathcal{S}_1^=$.

Case $[\![e]\!]_p = \mathsf{let}\, s = (\{r\} \cap p) \vee s \,\mathsf{in}\, [\![e']\!]_p$. In this case $e = \mathsf{enable}\, r \,\mathsf{in}\, e'$, and by Lemma 2.6 and Lemma 3.6 and $\forall$ ELIM we have subderivations of the following form, where $R = p \cap \{r\}$, $\varsigma = \{R\bar{\varphi}, \rho\}$ and $\varsigma' = \{R+, \rho\}$:

$$\dfrac{\Gamma' \vdash \vee R : \{R\bar{\varphi}, \rho\} \to \varsigma' \qquad \dfrac{\Gamma'(s) = \{R\bar{\varphi}, \rho\}}{\Gamma' \vdash s : \{R\bar{\varphi}, \rho\}}}{\Gamma' \vdash R \vee s : \varsigma'}$$

$$\dfrac{\dfrac{\Gamma' \vdash R \vee s : \varsigma' \qquad \bar{\alpha} \cap \mathrm{fv}(\Gamma') = \varnothing}{\Gamma' \vdash R \vee s : \forall \bar{\alpha}.\varsigma'} \qquad \Gamma'; s : \forall \bar{\alpha}.\varsigma' \vdash [\![e']\!]_p : \tau}{\Gamma' \vdash \mathsf{let}\, s = R \vee s \,\mathsf{in}\, [\![e']\!]_p : \tau}$$

However, since ς occurs unbound in Γ' so that any variables in ρ are free in Γ', and $\bar{\alpha} \cap \mathrm{fv}(\Gamma') = \varnothing$ by the above, therefore $\forall \bar{\alpha}.\varsigma'$ is dead and thus $\Gamma'; s : \varsigma' \vdash [\![e']\!]_p : \tau$ is derivable by Lemma 4.15. Therefore, since $(\Gamma'; s : \varsigma') = (\Gamma; s : \varsigma')$, we have that $p, \varsigma', \Gamma \vdash e' : \tau$ is derivable by the induction hypothesis, so that $p, \varsigma, \Gamma \vdash \mathsf{enable}\, r\, \mathsf{in}\, e' : \tau$ is derivable by ENABLE FAILURE if $r \notin p$, or ENABLE SUCCESS if $r \in p$.

Case $[\![e]\!]_p = \mathsf{let}\,\_ = s \ni r\, \mathsf{in}\, [\![e']\!]_p$. In this case $e = \mathsf{check}\, r\, \mathsf{then}\, e'$, and by Lemma 2.6 and definition of Δ_1 we have a subderivation of the following form, where $\varsigma = \{r+, \rho\}$:

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash \ni_r : \varsigma \to \varsigma \qquad \dfrac{\Gamma'(s) = \varsigma}{\Gamma' \vdash s : \varsigma}}{\Gamma' \vdash s \ni r : \varsigma} \qquad \bar{\alpha} \cap \mathsf{fv}(\Gamma') = \varnothing}{\Gamma' \vdash s \ni r : \forall \bar{\alpha}.\varsigma} \qquad \Gamma'; \_ : \forall \bar{\alpha}.\varsigma \vdash [\![e']\!]_p : \tau}{\Gamma' \vdash \mathsf{let}\,\_ = s \ni r\, \mathsf{in}\, [\![e']\!]_p :}$$

But since $\_$ does not occur in e', therefore by Lemma 4.16 we have that $\Gamma' \vdash [\![e']\!]_p : \tau$ is derivable such that $p, \varsigma, \Gamma \vdash e' : \tau$ is derivable by the induction hypothesis. Thus, $p, \varsigma, \Gamma \vdash \mathsf{check}\, r\, \mathsf{then}\, e' : \tau$ is derivable by CHECK.

Case $[\![e]\!]_p = ?_s\, r\, (\lambda s.[\![e_1]\!]_p)\, (\lambda s.[\![e_2]\!]_p)$. In this case $e = \mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ and by Lemma 2.6 and definition of Δ_1 we have subderivations of the following form, where $\varsigma = \{r\varphi, \rho\}$:

$$\dfrac{\Gamma' \vdash ?_r : \{r\varphi, \rho\} \to (\{r+, \rho\} \to \tau) \to (\{r-, \rho\} \to \tau) \to \tau \qquad \Gamma' \vdash s : \{r\varphi, \rho\}}{\Gamma' \vdash ?_s\, r : (\{r+, \rho\} \to \tau) \to (\{r-, \rho\} \to \tau) \to \tau}$$

$$\dfrac{\Gamma'; s : \{r+, \rho\} \vdash [\![e_1]\!]_p : \tau}{\Gamma' \vdash \lambda s.[\![e_1]\!]_p : \{r+, \rho\} \to \tau} \qquad \dfrac{\Gamma'; s : \{r-, \rho\} \vdash [\![e_1]\!]_p : \tau}{\Gamma' \vdash \lambda s.[\![e_2]\!]_p : \{r-, \rho\} \to \tau}$$

$$\dfrac{\Gamma' \vdash ?_s\, r : (\{r+, \rho\} \to \tau) \to (\{r-, \rho\} \to \tau) \to \tau \qquad \Gamma' \vdash \lambda s.[\![e_1]\!]_p : \{r+, \rho\} \to \tau}{\Gamma' \vdash ?_s\, r\, (\lambda s.[\![e_1]\!]_p) : (\{r-, \rho\} \to \tau) \to \tau}$$

$$\dfrac{\Gamma' \vdash ?_s\, r\, (\lambda s.[\![e_1]\!]_p) : (\{r-, \rho\} \to \tau) \to \tau \qquad \Gamma' \vdash \lambda s.[\![e_2]\!]_p : \{r-, \rho\} \to \tau}{\Gamma' \vdash ?_s\, r\, (\lambda s.[\![e_1]\!]_p)\, (\lambda s.[\![e_2]\!]_p) : \tau}$$

But since $(\Gamma'; s : \{r-, \rho\}) = (\Gamma; s : \{r-, \rho\})$, therefore by the induction hypothesis and the above we have that $p, \{r+, \rho\}, \Gamma \vdash e_1 : \tau$ and $p, \{r-, \rho\}, \Gamma \vdash e_2 : \tau$ are both derivable, so $p, \{r\varphi, \rho\}, \Gamma \vdash \mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : \tau$ is derivable by TEST.

Case $[\![e]\!]_p = \mathsf{let}\, s = p' \wedge s\, \mathsf{in}\, [\![e']\!]_{p'}$. In this case $e = p'.e'$, and by Lemma 2.6, Lemma 3.6

and $\forall$ ELIM we have subderivations of the following form, where $\varsigma = \{p'\bar{\varphi}, \rho\}$ and $\varsigma' = \{p'\bar{\varphi}, \varnothing\}$:

$$\dfrac{\Gamma' \vdash \wedge p' : \{p'\bar{\varphi}, \rho\} \to \varsigma' \qquad \dfrac{\Gamma'(s) = \{p'\bar{\varphi}, \rho\}}{\Gamma' \vdash s : \{p'\bar{\varphi}, \rho\}}}{\Gamma' \vdash p' \wedge s : \varsigma'}$$

$$\dfrac{\dfrac{\Gamma' \vdash p' \wedge s : \varsigma' \qquad \bar{\alpha} \cap \mathsf{fv}(\Gamma') = \varnothing}{\Gamma' \vdash p' \wedge s : \forall\bar{\alpha}.\varsigma'} \qquad \Gamma'; s : \forall\bar{\alpha}.\varsigma' \vdash [\![e']\!]_{p'} : \tau}{\Gamma' \vdash \mathsf{let}\, s = p' \wedge s \,\mathsf{in}\, [\![e']\!]_{p'} : \tau}$$

However, since ς occurs unbound in Γ' so that any variables in $\bar{\varphi}$ are free in Γ', and $\bar{\alpha} \cap \mathrm{fv}(\Gamma') = \varnothing$ by the above, therefore $\forall\bar{\alpha}.\varsigma'$ is dead and thus $\Gamma'; s : \varsigma' \vdash [\![e']\!]_p : \tau$ is derivable by Lemma 4.15. Therefore, since $(\Gamma'; s : \varsigma') = (\Gamma; s : \varsigma')$, we have that $p', \varsigma', \Gamma \vdash e' : \tau$ is derivable by the induction hypothesis, so that $p, \varsigma, \Gamma \vdash p'.e' : \tau$ is derivable by SIGN. □

Two more utility Lemmas to handle the details of the top-level λ_{sec}-to-pml_B transformation $(\!| e |\!)$, and then the desired correspondence result:

Lemma 4.19 *If $C, \Gamma; x : \sigma' \vdash e : \sigma$ and $C, \Gamma \vdash v : \sigma'$ are derivable then so is $C, \Gamma \vdash e[v/x] : \sigma$.*

Lemma 4.20 *If $C, \Gamma \vdash e[R/x] : \sigma$ and $C, \Gamma \vdash R : \varsigma$ is derivable, then so is $C, (\Gamma; x : \varsigma) \vdash e : \sigma$.*

Theorem 4.5 *The judgment nobody, $\{\varnothing\}, \Gamma \vdash e : \tau$ is derivable in $\mathcal{S}_1^=$ iff* **true**, $(\Gamma; s : \{\varnothing\}) \vdash (\!| e |\!) : \tau$ *is derivable in HM*(RS$^=$).

Proof. Suppose on the one hand that *nobody*, $\{\varnothing\}, \Gamma \vdash e : \tau$ is derivable. By Lemma 4.14 we have that **true**, $(\Gamma; s : \{\varnothing\}) \vdash [\![e]\!]_{\textit{nobody}} : \tau$ is derivable, so that by Lemma 4.19 we have that **true**, $(\Gamma) \vdash (\!| e |\!) : \tau$ is derivable, since $(\!| e |\!) = [\![e]\!]_{\textit{nobody}}[\varnothing/s]$ and **true**, $(\Gamma) \vdash \varnothing : \{\varnothing\}$ is derivable by definition of Δ_1 and CONST.

Suppose on the other hand that **true**, $(\Gamma) \vdash (\!| e |\!) : \tau$ is derivable. Therefore, since $(\!| e |\!) = [\![e]\!]_{\textit{nobody}}[\varnothing/s]$ and **true**, $(\Gamma) \vdash \varnothing : \{\varnothing\}$ by definition of Δ_1 and CONST, we have that **true**, $(\Gamma; s : \{\varnothing\}) \vdash [\![e]\!]_{\textit{nobody}} : \tau$ is derivable by Lemma 4.20. But then by Lemma 4.18 we have that *nobody*, $\{\varnothing\}, \Gamma \vdash e : \tau$ is derivable. □

Given this correspondence and Theorem 4.3, progress and type safety results for λ_{sec} in the direct $\mathcal{S}_1^=$ system are immediate:

Theorem 4.6 (λ_{sec} Progress) *If e is well-typed then either $e \to^{\star} v$ or e diverges.*

Theorem 4.7 (λ_{sec} Type Safety) *If e is well-typed then e does not go wrong.*

Furthermore, we note that by these results and Theorem 4.2, Theorems 1.2 and 1.1 follow, since the language and type system presented in this Chapter subsume those presented in Chapter 1.

Another important consequence of these results is that, similarly to Proposition 1.1, we may now formally assert that runtime stack inspection for privilege checks is no longer necessary, which follows immediately by Theorem 4.7:

Corollary 4.1 (λ_{sec} Optimization) *Let $\rightsquigarrow$ be defined as $\to$, but with the rule:*

$$E[\mathsf{check}\, r\, \mathsf{then}\, e] \quad \to \quad E[e] \quad \textit{if } E \vdash r$$

replaced with:

$$E[\mathsf{check}\, r\, \mathsf{then}\, e] \quad \rightsquigarrow \quad E[e]$$

and suppose e is well-typed; then $e \rightsquigarrow^{\star} v$ iff $e \to^{\star} v$.

This result states that runtime stack inspection in programs that contain only privilege checks may be eliminated entirely. Note, however, that this result says nothing about runtime checks performed in the case of privilege tests. In fact, recalling the initial bindings in Fig. 3.9, the systems $\mathcal{S}_1^{rel}$ do not have precise enough types for eliminating run-time stack inspection for privilege tests. The systems $\mathcal{S}_1^{rel}$ do, but the mechanism for doing so would be more complicated, involving the trimming of branches which are statically known, by the type conditions, to be unfollowed. It is not clear what balance of type precision and run-time tests would be most effective in practice, remaining an interesting topic for future work.

4.4.4 Type inference

Type inference for λ_{sec} can be obtained in the same manner as the logical type systems. *Indirect* type inference may be defined as the composition of the λ_{sec}-to-pml_B transformation with any of the pml_B type inference methods discussed in Sect. 3.3.5. *Direct* type inference for λ_{sec} may be derived from indirect inference in the same manner that direct judgements are derived from the indirect.

In the Appendix, a direct type inference algorithm for λ_{sec} in $\mathcal{S}_1^{=}$ is defined in the module `Typing`, specifically in the functor `System`. This functor is parameterized by a module `C`

: `Context`, which specifies a local access control list and initial principal; an example of such a module is given in `LocalContext`. This type inference algorithm uses exactly the same constraint system and normalization procedure `unify` employed for pml_B $\mathcal{S}_1^=$ type inference. Proving this direct inference algorithm correct would then be a straightforward manner of showing a correspondence between the syntax-directed inference rules in modules `Hmx` and `Typing`. The OCaml implementation of the λ_{sec} constraint system and `unify` procedure are not included in the Appendix, but are available online at `http://www.cs.jhu.edu/~ces/thesis/impl/direct`.

4.5 Examples and discussion

In this section, we give examples which illustrate the expressivity (and limitations) of our type system. These examples allow discussing the differences between the variants of the system, yielding insights into the possible trade-offs between precision and cost.

4.5.1 Security wrappers

A library writer often needs to surround numerous internal functions with "boilerplate" security code before making them accessible. To avoid redundancy, it seems desirable to allow the definition of generic *security wrappers*. When applied to a function, a wrapper returns a new function which has the same computational meaning but different security requirements. Assume given a principal $p = \{r, s\}$; here are two security wrappers likely to be useful to programmers:

$$\mathit{enable}_r \triangleq \lambda f.p.\lambda x.p.\textsf{enable}\, r \,\textsf{in}\, f\, x$$
$$\mathit{require}_r \triangleq \lambda f.p.\lambda x.p.\textsf{check}\, r \,\textsf{then}\, f\, x$$

In system $\mathcal{S}_1^=$, these wrappers receive the following (most general) types:

$$\mathit{enable}_r : \forall \ldots.(\alpha_1 \xrightarrow{\{r+, s\gamma_1, \varnothing\}} \alpha_2) \xrightarrow{\{\beta_1\}} (\alpha_1 \xrightarrow{\{r\gamma_2, s\gamma_1, \beta_2\}} \alpha_2)$$
$$\mathit{require}_r : \forall \ldots.(\alpha_1 \xrightarrow{\{r+, s\gamma_1, \varnothing\}} \alpha_2) \xrightarrow{\{\beta_1\}} (\alpha_1 \xrightarrow{\{r+, s\gamma_1, \beta_2\}} \alpha_2)$$

These types are very similar; they may be read as follows. Both wrappers expect a function f which allows that r be enabled ($r+$), i.e. one which *either* requires r to be enabled, *or* doesn't care about its status. (Indeed, as in ML, the type of the actual argument may be more general than that of the formal.) They return a new function with identical domain and codomain (α_1, α_2), which works regardless of r's status (enable_r yields $r\gamma_2$) or requires r to be enabled ($\mathit{require}_r$ yields $r+$). The new function retains f's expectations about s ($s\gamma_1$). f must not require any further privileges ($\varnothing$), because it is invoked by p, which enjoys privileges r and s only.

These polymorphic types are very expressive. Our main concern is that, even though the privilege s is not mentioned in the *code* of these wrappers, it does appear in their *type*. More generally, every privilege available to p may show up in the type of a function written on behalf of principal p, which may lead to very verbose types. An appropriate type abbreviation mechanism may be able to address this problem; this is left as a subject for future work.

4.5.2 Use and types of security tests

This example displays two typical programming idioms involving test. One (arguably the most common) is very simple, and may be typed in $\mathcal{S}_1^=$. The other is more complex and requires at least $\mathcal{S}_2^=$. We take this opportunity to discuss various problems related to the interpretation of conditional constraints.

Imagine an operating system with two kinds of processes, root processes and user processes. Killing a user process is always allowed, while killing a root process requires the privilege *killing*. At least one distinguished principal *root* has this privilege. The system functions which perform the killing are implemented by *root* as follows, assuming the trivial addition of a unit value and type to λ_{sec}:

$$\begin{aligned} \mathit{kill} &\triangleq \lambda(p:\mathit{process}).\mathit{root}.\mathsf{check}\ \mathit{killing}\ \mathsf{then}\ \ldots\ () \quad \text{– \textit{kill the process}} \\ \mathit{killIfUser} &\triangleq \lambda(p:\mathit{process}).\mathit{root}.\ldots\ () \quad \text{– \textit{kill the process if it is user-level}} \end{aligned}$$

In system $\mathcal{S}_1^=$, these functions receive the following (most general) types:

$$\begin{aligned} \mathit{kill} &: \forall\beta.\mathit{process} \xrightarrow{\{\mathit{killing}+,\beta\}} \mathit{unit} \\ \mathit{killIfUser} &: \forall\beta.\mathit{process} \xrightarrow{\{\beta\}} \mathit{unit} \end{aligned}$$

The first function can be called only if it can be statically proven that the privilege *killing* is enabled. The second one, on the other hand, can be called at any time, but will never kill a root process. To complement these functions, it may be desirable to define a function which provides a "best attempt" given the current (dynamic) security context. This may be done by dynamically checking whether the privilege is enabled, then calling the appropriate function:

$$\begin{aligned} &\mathit{tryKill} \triangleq \lambda(p:\mathit{process}).\mathit{root}. \\ &\qquad\qquad \mathsf{test}\ \mathit{killing}\ \mathsf{then}\ \mathit{kill}(p)\ \mathsf{else}\ \mathit{killIfUser}(p) \end{aligned}$$

This function is well-typed in system $\mathcal{S}_1^=$. Indeed, within the first branch of the test construct, it is statically known that the privilege *killing* must be enabled; this is why the sub-expression $\mathit{kill}(p)$ is well-typed. The inferred type shows that *tryKill* does not have any security requirements:

$$\mathit{tryKill} : \forall\beta.\mathit{process} \xrightarrow{\{\beta\}} \mathit{unit}$$

In the function *tryKill*, the sensitive action *kill* is performed within the lexical scope of the test construct, which is why it is easily seen to be safe. However, one can also move it outside of the scope, as follows:

$$\begin{array}{rcl} \mathit{tryKill'} & \triangleq & \lambda(p : \mathit{process}).\mathit{root}. \\ & & \quad \mathsf{let}\, \mathit{action} = \mathsf{test}\, \mathit{killing}\, \mathsf{then}\, \mathit{kill}\, \mathsf{else}\, \mathit{killIfUser}\, \mathsf{in}\, \mathit{action}\, p \end{array}$$

Here, the dynamic security check yields a closure, whose behavior depends on the check's outcome. It can be passed on and used in further computations. Such a programming idiom is useful in practice, because it allows hoisting a security check out of a loop. For instance, if we were to kill a set of processes, instead of a single one, we would apply *action* successively to each element of the set. Thus, only one security check would have to be performed, regardless of the number of processes in the set.

Is *tryKill'* also well-typed? This is more subtle. In those $\mathcal{S}_i^{rel}$ where $i = 1$, the two branches of a test construct must receive the same type. Because the function *kill* requires a non-trivial security context, it is conservatively assumed that *action* may do so as well. As a result, in e.g. $\mathcal{S}_1^=$, *tryKill'* has the following (most general) type:

$$\mathit{tryKill'} : \forall\beta.\mathit{process} \xrightarrow{\{\mathit{killing}+,\beta\}} \mathit{unit}$$

which is the same as *kill*'s type. Thus, it is well-typed, but its type is more restrictive than expected.

To solve this problem, we need to keep track of the fact that the behavior (i.e. the type) of *action*depends on the outcome of the check, i.e. on whether the privilege *kill* is enabled. This is precisely the reason for moving to the column $i = 2$ in our array of type systems. In this column, the result of a test construct is described by conditional constraints, which encode the desired dependency. Indeed, in $\mathcal{S}_2^=$, *tryKill'* has the following (most general) type:

$$\begin{array}{rl} \mathit{tryKill'} : & \forall\beta[C].\mathit{process} \xrightarrow{\{\mathit{killing}\gamma,\beta\}} \alpha \\ \text{where } C = & \ \text{if } + \leq \gamma \text{ then } \mathit{unit} \leq \alpha \\ & \wedge\ \text{if } - \leq \gamma \text{ then } \mathit{unit} \leq \alpha \end{array}$$

This type no longer requires the privilege *kill* to be enabled: our analysis was smart enough to prove that this code is safe.

The reader may wonder why we haven't unified α with *unit*, since both $\gamma = +$ and $\gamma = -$ imply *unit* $= \alpha$. This is because there remain other cases (namely $\gamma = \bot$ and $\gamma = \top$) where α is unconstrained; as a result, these conditional constraints do not logically imply *unit* $= \alpha$. To fix this apparent problem, it would be possible to remove $\bot$ and $\top$ from the model. In that case, imposing *unit* $= \alpha$ would be a valid simplification. However, this would make the constraint satisfaction problem much more complex – we conjecture, exponential. To see why, notice that the system would then be powerful enough to express disjunctive types. Indeed, the type $\tau_1 \vee \tau_2$ would be expressible as a type variable α accompanied with the constraints

$$\text{if } + \leq \beta \text{ then } \tau_1 \leq \alpha$$

$$\text{if } - \leq \beta \text{ then } \tau_2 \leq \alpha$$

(where α and β are fresh). The fact that β must be equal to either $+$ or $-$ (because there are no other elements in the model) means that α must be equal to τ_1 or τ_2. Implementing a constraint solver which does not naïvely try both cases separately seems problematic.

Another interesting possibility consists in giving a different interpretation to conditional constraints. Notice that we really wish to use conditional constraints in application to privilege `test`s in only a very limited way. We want to allow the branches of a `test` construct to receive different types— but we do not wish for these types to differ an *arbitrary* ways; we only wish to allow their *security annotations* to differ. Doing so turns out to be very easy, at least from a purely theoretical point of view. Define $\approx$ as the binary relation which is uniformly true on $[\![Row(c)_{\varnothing}]\!]$, and extend it as a straightforward equivalence to $[\![k]\!]$ for every kind k. Then, re-define the interpretation of simple conditional constraints as follows:

$$\frac{\rho(\tau') \approx \rho(\tau'') \qquad c \leq \rho(\tau) \Rightarrow \rho \vdash \tau' \leq \tau''}{\rho \vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''}$$

This interpretation requires the types which appear in the conclusion of a conditional constraint (here, τ' and τ'') to be equal modulo security annotations. This allows the *structure* of types to be determined using rigid rules (which is desirable, because many programming errors are then detected earlier), while keeping the flexibility of conditional reasoning on security annotations. Under such an interpretation, the $\mathcal{S}_2^=$ type of *tryKill'* specified above is logically equivalent to the following, as desired:

$$\textit{tryKill'} : \forall\beta.\textit{process} \xrightarrow{\{\beta\}} \textit{unit}$$

4.5.3 Subtyping

All of the examples given so far can be given useful types in $\mathcal{S}_i^=$ for some $i \in \{1, 2\}$. In other words, these examples do not require subtyping. Nevertheless, there are a few cases where the extra precision afforded by subtyping becomes necessary.

Imagining the straightforward addition of a conditional construct to the language, suppose that we write a slightly modified version of the wrapper *enable*$_r$ presented in Sect. 4.5.1 as follows, where P is some condition:

$$\mathit{maybeEnable}_r \quad \triangleq \quad \lambda f.p.\lambda x.p.\,\mathsf{if}\, P\,\mathsf{then}\, f\,x\,\mathsf{else}\,\mathsf{enable}\, r\,\mathsf{in}\, f\,x$$

This wrapper may or may not enable the privilege r before calling f. In $\mathcal{S}_i^=$, its (most general) type is as follows:

$$\mathit{maybeEnable}_r : \forall \ldots.(\alpha_1 \xrightarrow{\{r+, s\gamma_1, \varnothing\}} \alpha_2) \xrightarrow{\{\beta_1\}} (\alpha_1 \xrightarrow{\{r+, s\gamma_1, \beta_2\}} \alpha_2)$$

But this is exactly the same as the type of *require*$_r$ specified in Sect. 4.5.1— in other words, the type system thinks application of *maybeEnable*$_r$ yields a function that requires the privilege r! How was such an overly conservative conclusion drawn?

The cause of this imprecision is unification together with the restrictions inherent in let-polymorphism. Because f is λ-bound (not let-bound), all of its uses must receive the same type, say $\alpha_1 \xrightarrow{\varsigma} \alpha_2$. In the second branch of the if statement, f is called with r enabled; thus, ς must be of the form $\{r+, \ldots\}$ within that branch. In the first branch of the if statement, f is called within an unmodified security context. The type system then concludes that the wrapped function has the same security requirement $\{r+, \ldots\}$ in both branches, as a result of our use of equality constraints and unification— because f *may* be called with r enabled, this leads us to believe f *must* be called with r enabled.

One standard solution is to move to a system where equality is replaced with subtyping, e.g. $\mathcal{S}_1^\leq$. There, we may obtain the following type for *maybeEnable*$_r$:

$$\begin{gathered}\mathit{maybeEnable}_r : \forall \ldots [C].(\alpha_1 \xrightarrow{\{r\gamma, s\gamma_1, \varnothing\}} \alpha_2) \xrightarrow{\{\beta_1\}} (\alpha_1 \xrightarrow{\{r\gamma_2, s\gamma_1, \beta_2\}} \alpha_2)\\ \text{where } C = + \leq \gamma \wedge \gamma_2 \leq \gamma\end{gathered}$$

This type is much more permissive, because $\gamma_2 \leq \gamma \geq +$ does not allow concluding $\gamma_2 \leq +$ (as is the case when $\leq$ is interpreted by equality). Indeed, it may be that γ_2 assumes the type $-$, i.e. application of *maybeEnable*$_r$ yields a function that requires r to be disabled. The constraint $+ \leq \gamma \wedge \gamma_2 \leq \gamma$ then requires $\top \leq \gamma$, i.e. f must be able to accept either state of the privilege r.

Our experience seems to indicate that subtyping is useful only when polymorphism is inhibited, i.e. when using higher-order functions. Java has no such construct. Java does have first-class objects, which contain methods, but it seems reasonable to require that methods be given explicit polymorphic types by the user as part of class declarations. Considering that subtyping has substantial cost in terms of readability and efficiency, it may then be interesting *not* to use it in a real-world system. However, more work is needed to confirm this conjecture.

Chapter 5

Types for Object Confinement

In this Chapter we switch gears a bit, turning our attention to a different language-based security model— object confinement, aka capability-based security for OO languages. However, while the security model is different, we will use the same *transformational* technique to develop a static analysis for the language. Somewhat surprisingly, we will in fact be able to use the same target language for the transformation studied here— the language pml_B defined in Chapter 2— as that used for the λ_{sec} transformation in Chapter 4. The benefits of this transformational approach will again be a significantly reduced proof effort for demonstrating type safety, while the benefits of a static analysis will again include the possibility of run-time optimizations, and a clearer declaration of security policies.

The confinement of object references is a significant security concern in languages such as Java. Aliasing and other features of OO languages can make this a difficult task; recent work [43, 5] has focused on the development of type systems for enforcing various containment policies in the presence of these features. In this chapter, we describe a new language and type system for the implementation of object confinement mechanisms that is more general than previous systems, and which is based on a different notion of security enforcement.

Object confinement is closely related to *capability*-based security, utilized in several operating systems such as EROS [36], and also in programming language (PL) architectures such as J-Kernel [13], E [8], and Secure Network Objects [42]. A capability can be defined as a reference to a data segment, along with a set of access rights to the segment [17]. An important property of capabilities is that they are *unforgeable*: it cannot be faked or reconstructed from partial information. In Java, object references are likewise unforgeable, a property enforced by the type system; thus, Java can also be considered a statically enforced capability system.

So-called *pure* capability systems rely on their high level design for safety, without any additional system-level mechanisms for enforcing security. Other systems *harden* the pure model by layering other mechanisms over pure capabilities, to provide stronger system-level enforcement of security; the `private` and `protected` modifiers in Java are an example of this. Types improve the hardening mechanisms of capability systems, by providing a declarative statement of security policies, as well as improving run-time efficiency through static, rather than dynamic, enforcement of security. Our language model and static type analysis focuses on capability hardening, with enough generality to be applicable to a variety of systems, and serves as a foundation for studying object protection in OO languages.

5.1 Overview of the pop system

In this section, we informally describe some of the ideas and features of our language, called pop, and show how they improve upon previous systems. As will be demonstrated in Sect. 5.5, pop is sufficient to implement various OO language features, e.g. classes with methods and instance variables, but with stricter and more reliable security.

Use vs. communication-based security

Our approach to object confinement is related to previous work on containment mechanisms [2, 43, 5], but has a different basis. Specifically, these containment mechanisms rely on a *communication*-based approach to security; some form of barriers between objects, or domain boundaries, are specified, and security is concerned with communication of objects (or object references) across those boundaries. In our *use*-based approach, we also specify domain boundaries, but security is concerned with how objects are *used* within these boundaries. Practically speaking, this means that security checks on an object are performed when it is used (selected), rather than communicated.

The main advantage of the use-based approach is that security specifications may be more fine-grained; in a communication based approach we are restricted to a whole-object "what-goes-where" security model, while with a use-based approach we may be more precise in specifying what methods of an object may be used within various domains. This is particularly relevant to access control. Use-based security also more closely corresponds to traditional capability-based security models in practice, where capabilities are not just references, but are references plus an interface specifying access rights.

In addition, our use-based security model allows "tunneling" of objects: a capability may pass through a domain where its use is disallowed, provided it is not used in that domain. This supports the multitude of protocols which rely on an intermediary that is not fully trusted. In a communication-based model capabilities are prevented from passing through unauthorized domains, so tunneling is impossible.

Static protection domains

The pop language is an object-based calculus, where object methods are defined by lists of method definitions in the usual manner. For example, substituting the notation $\ldots$ for the syntactic details, the definition of a file object with read and write methods would appear as follows:

$$[\mathrm{read}() = \ldots, \mathrm{write}(x) = \ldots] \cdot \ldots \cdot \ldots$$

Additionally, every object definition statically asserts membership in a specific *protection domain* d, so that expanding on the above we could have:

$$[\mathrm{read}() = \ldots, \mathrm{write}(x) = \ldots] \cdot d \cdot \ldots$$

While the system requires that all objects are annotated with a domain, the *meaning* of these domains is flexible, and open to interpretation. Our system, considered in a pure form, is a core analysis that may be specialized for particular applications. For example, domains may be as interpreted code owners, or they may be interpreted as denoting regions of static scope—e.g. package or object scope.

Along with domain labels, the language provides a method for specifying a security policy, dictating how domains may interact, via *user interface* definitions φ. Each object is annotated with a user interface, so that letting φ be an appropriately defined user interface and again expanding on the above, we could have:

$$[\mathrm{read}() = \ldots, \mathrm{write}(x) = \ldots] \cdot d \cdot \varphi$$

We describe user interfaces more precisely below, and illustrate and discuss relevant examples in Sect. 5.5. For now, we note that the flexibility in the interpretation of domains implies a flexibility in the style of policies that may be enforced: e.g. if domains are interpreted as code-owner labels, then the policy is access control, while if domains are interpreted as static scope, then the policy is a use-based access modifier mechanism.

Object interfaces

Other secure capability-based language systems have been developed [13, 42, 8] which include a notion of an access-rights interface, in the form of object types. Our system provides a more fine-grained mechanism: for any given object, its user-interface definition φ may be defined so that different domains are given more or less restrictive views of the same object, and these views are statically enforced. Note that the use-based, rather than communication-based approach to security is an advantage here, since the latter allows us to more precisely modulate *how* an object may be used by different domains, via object method interfaces.

For example, we can imagine that any object in domain d is a "friend" and should be given free reign over other objects in d, whereas objects in domain d' are somewhat trusted but potentially hostile, so that we might wish such objects to read data in d but not be able to alter it. Thus, returning to our previous example, an appropriate definition of φ in the file object definition, given these security preconceptions, would be as follows:

$$[\mathrm{read}() = \ldots, \mathrm{write}(x) = \ldots] \cdot d \cdot \left\{d \mapsto \{\mathrm{read}, \mathrm{write}\}, d' \mapsto \{\mathrm{read}\}\right\}$$

User interfaces may additionally contain mappings for a *default* user ∂, which allows the programmer to specify interfaces for domains which may not be known at compile time. Thus, the system allows for a degree of "open-endedness" in its design. Returning to the previous example, if our policy was to allow *any* domain read access to files in domain d, we could define files and associated interfaces in that domain as follows:

$$[\mathrm{read}() = \ldots, \mathrm{write}(x) = \ldots] \cdot d \cdot \{d \mapsto \{\mathrm{read}, \mathrm{write}\}, \partial \mapsto \{\mathrm{read}\}\}$$

The notation ∂ matches any domain. As is the case for normal interface specifications, the access rights associated with default interfaces are statically enforced.

The user interface is a mapping from domains to access rights—that is, to sets of methods in the associated object that each domain is authorized to use. This looks something like an ACL-based security model; however, ACLs are defined to map *principals* to privileges. Domains, on the other hand, are fixed boundaries in the code which may have nothing to do with principals. The practical usefulness of a mechanism with this sort of flexibility has been described in [4], in application to mobile programs. Other applications and more detailed examples are discussed in Sect. 5.5, including an encoding of `private` and `protected` method and instance variable modifiers.

Weak capabilities

The EROS weak capabilities mechanism, described in [36], allows an enforcement of "transitive read-only" properties via a sort of deep-casting mechanism. We model weak capabilities in the system presented here, and statically enforce weakening properties via types. In fact, we provide a generalization of the EROS conception of weak capabilities.

In EROS, capabilities are low-level entities which may possess a fixed number of primitive access rights such as read and write. A weakened capability is read only, and any capabilities read from a weakened capability are automatically weakened. In our higher-level system, capabilities are objects, with access rights corresponding to the user-defined methods in these objects; our weakening mechanism is similarly generalized to apply to any method access rights. This generalization of the EROS weakening mechanism is particularly useful in the realm of recursively defined object structures. For example, it can be used to enforce recursive read-only properties in a filesystem where files may contain other filehandles, or recursively disable delete permissions throughout a directory tree. We now elaborate on this latter example.

If o is a directory object and delete is a directory object method that allows deletion of entities in a directory, then the expression

$$\mathbf{weak}_{\{\text{delete}\}}(o)$$

denotes a weakening of o such that deletion within that directory is disallowed, and furthermore, if $m \neq$ delete is an accessible method of o which returns another directory object o', then o' will be similarly weakened to disallow deletion. The type system statically enforces this mechanism in a flexible manner. A concrete example of weakened read-write cell definitions, along with type system enforcement of these properties, is given in Sect. 5.5.

Casting

We also provide a *casting* mechanism, that allows removal of access rights from particular views of an object, allowing a greater attenuation of security when necessary. Again, this casting discipline is statically enforced. For example, letting o be the pop object defined immediately above, if some circumstance suggests that we should no longer allow objects in domain d' read access to files in d, then we may make the following cast:

$$o \upharpoonright (d', \varnothing)$$

This removes all of d' access privileges on o, by setting the set of d''s accessible methods to $\varnothing$. Significantly, we allow only "upcasts", so that privileges can be removed, but never added.

Type systems already have a built-in notion of interface and of restriction of interfaces, via subtyping and subsumption. Our system is inspired by and sits on a foundation of subtyping, but is significantly more general. Most importantly, privileges can be restricted by subtyping in standard systems, but this is only with respect to two implicit domains: the local one and everything else. With our explicit domains and fine-grained user interface definitions, casting restrictions may be significantly more fine-grained, as seen in the previous example.

Rights amplification

Capability-based security systems support several forms of *rights amplification*, the temporary and disciplined amplification of rights in certain program contexts. One form of rights amplification is by indirection. obtained, it For example, letting o be the file object as defined immediately above, and recalling that d was the only domain allowed write access to o, we may allow another object in domain d to function as a write-access "proxy" to o, as in the following definition:

$$[\mathrm{proxywrite}(x) = o.\mathrm{write}(x)] \cdot d \cdot \{\partial \mapsto \{\mathrm{proxywrite}\}\}$$

Any object in any domain may use this object to gain write-access to o, though *direct* write-access to o is restricted. This example is extreme and not a recommended programming style, but a limited use "by proxy" of capabilities not directly held is a common idiom in capability-based programming. This must also must be kept in mind when a capability is doled out—the doler must be aware of both direct and indirect actions allowed by it. In contrast, the stack-inspection security model discussed in previous chapters allows restrictions to be placed on what an invoker can induce in an invoked object—if the original invoker had no access rights, this is recorded on the stack and access can be prevented. This is one of the most significant differences between the stack-inspection and capability-based security models.

5.2 The pop language definition

We now formally define the syntax and operational semantics of pop, an object-based language with state and capability-based security features, described informally in the previous sections.

$$
\begin{array}{llll}
m, x, \mathbf{s}, \mathrm{set}, \mathrm{get} \ \in \ \mathit{ID}, \ \iota \subseteq \mathit{ID} & & & \textit{identifiers} \\
l \in \mathit{Loc} & & & \textit{locations} \\
d \in \mathcal{D}, \ D \subseteq \mathcal{D} & & & \textit{domains} \\
\varphi \in \mathcal{D} \to 2^{\mathit{ID}} & & & \textit{interfaces} \\
\varrho & ::= & \overline{m_i(x) = e_i}^{\,0<i\leq n} & \textit{method lists} \\
s & ::= & [\varrho] & \textit{self objects} \\
co & ::= & [\varrho] \cdot d \mid l & \textit{core objects} \\
o & ::= & co \cdot \varphi\backslash_\iota \mid s & \textit{objects} \\
v & ::= & x \mid o & \textit{values} \\
e & ::= & v \mid e.m(e) \mid e \upharpoonright (d, \iota) \mid \mathbf{weak}_\iota(e) \mid \mathsf{let}\, x = v \,\mathsf{in}\, e \mid \mathrm{ref}_\varphi e \mid \cdot e \cdot & \textit{expressions} \\
E & ::= & [\,] \mid E.m(e) \mid v.m(E) \mid E \upharpoonright (d, \iota) \mid \mathrm{ref}_\varphi E \mid \mathbf{weak}_\iota(E) \mid \cdot E \cdot & \textit{evaluation contexts}
\end{array}
$$

Figure 5.1: Grammar for pop

5.2.1 Syntax

The grammar for pop is defined in Fig. 5.1. It includes a countably infinite set of identifiers $\mathcal{D}$ which we refer to as *protection domains*. The definition also includes the the following notation for method lists ϱ:

$$(\overline{m_i(x) = e_i}^{\,0<i\leq n}) \triangleq (m_1(x) = e_1, \ldots, m_n(x) = e_n)$$

Henceforth we will use a similar vector abbreviation notation for all language forms, with obvious meaning. We write $(m(x) = e) \in \varrho$ iff ϱ is of the form $(\ldots, m(x) = e, \ldots)$. Read-write cells are defined as primitives, with a cell constructor $\mathrm{ref}_\varphi v$ that generates a read-write cell containing v, with user interface φ. The object weakening mechanism $\mathbf{weak}_\iota(o)$, described in the previous section, is also provided.

Objects definitions are of the form $[\varrho] \cdot d \cdot \varphi\backslash_\iota$, where ι carries any methods removed by weakening. For convenience, and to retrieve the notation presented in the previous section, we define the syntactic sugar $(co \cdot \varphi) \triangleq (co \cdot \varphi\backslash_\varnothing)$. Self objects $[\varrho]$ are run-time entities, the dynamic implementation of self, and are disallowed in top-level programs.

User interfaces φ are *total* mappings from domain identifiers to sets of method names. Since they are user-defined in programs, the following syntactic sugar is provided, allowing a finite

$$
\begin{array}{rcll}
d :: \delta, ([\varrho] \cdot d' \cdot \varphi\backslash_\iota).m(v), \sigma & \hookrightarrow & d' :: d :: \delta, \cdot(\mathbf{weak}_\iota(e[[\varrho]/\mathbf{s}][v/x]))\cdot, \sigma & (\textit{send}) \\
& & \quad \text{if } (m(x) = e) \in \varrho \text{ and } m \in (\varphi(d, \partial)\backslash\iota) & \\
\delta, [\varrho].m(v), \sigma & \hookrightarrow & \delta, e[[\varrho]/\mathbf{s}][v/x], \sigma \qquad \text{if } (m(x) = e) \in \varrho & (\textit{self}) \\
\delta, (co \cdot \varphi\backslash_\iota) \shortmid (d', \iota'), \sigma & \hookrightarrow & \delta, (co \cdot (\varphi[d' \mapsto \iota'])\backslash_\iota), \sigma \qquad \iota' \subseteq \varphi(d', \partial) & (\textit{cast}) \\
\delta, \mathbf{weak}_\iota(co \cdot \varphi\backslash_{\iota'}), \sigma & \hookrightarrow & \delta, co \cdot \varphi\backslash_{(\iota \cup \iota')}, \sigma & (\textit{weaken}) \\
\delta, \text{ref}_\varphi v, \sigma & \hookrightarrow & \delta, l \cdot \varphi\backslash_\varnothing, \sigma[l \mapsto v] \qquad l \notin \text{dom}(\sigma) & (\textit{newcell}) \\
d :: \delta, (l \cdot \varphi\backslash_\iota).\text{set}(v), \sigma & \hookrightarrow & d :: \delta, \mathbf{weak}_\iota(v), \sigma[l \mapsto v] & (\textit{set}) \\
& & \quad l \in \text{dom}(\sigma) \text{ and set} \in (\varphi(d, \partial)\backslash\iota) & \\
d :: \delta, (l \cdot \varphi\backslash_\iota).\text{get}(), \sigma & \hookrightarrow & d :: \delta, \mathbf{weak}_\iota(\sigma(l)), \sigma \qquad \text{get} \in (\varphi(d, \partial)\backslash\iota) & (\textit{get}) \\
\delta, \mathsf{let}\, x = v \,\mathsf{in}\, e, \sigma & \hookrightarrow & \delta, e[v/x], \sigma & (\textit{let}) \\
d :: \delta, \cdot v \cdot, \sigma & \hookrightarrow & \delta, v, \sigma & (\textit{pop}) \\
\delta, E[e], \sigma & \rightarrow & \delta', E[e'], \sigma' \qquad \text{if } \delta, e, \sigma \hookrightarrow \delta', e', \sigma' & (\textit{context})
\end{array}
$$

Figure 5.2: Operational semantics for pop

specification of interfaces by implicitly mapping unspecified domains to $\varnothing$:

$$\left\{ d_i \mapsto \iota_i \ ^{0<i\leq n} \right\} \triangleq \left\{ d_i \mapsto \iota_i \ ^{0<i\leq n}, d_{i+1} \mapsto \varnothing, \ldots \right\}$$

We require that for any φ and d, the method names $\varphi(d)$ are a subset of the method names in the associated object. Note that object method definitions may contain the distinguished identifier $\mathbf{s}$ which denotes *self*, and which is bound by the scope of the object; objects always have full access to themselves via the identifier $\mathbf{s}$. We require that self never appear "bare"—that is, the variable $\mathbf{s}$ must always appear in the context of a method selection $\mathbf{s}.m(e)$. This restriction ensures that $\mathbf{s}$ cannot escape its own scope, unintentionally providing a "back-door" to the object. Self, and associated semantics, is discussed more thoroughly below.

5.2.2 Operational semantics

The small-step operational semantics for pop is defined in figure 5.2 as the relation $\rightarrow$ on *configurations* δ, e, σ, where *stores* σ are partial mapping from locations l to values v, and δ is a non-empty *domain stack*, the top element of which is called the *current domain*. Notation and language relevant to domain stacks is defined follows:

Definition 5.1 *Domain stacks are inductively defined as:*

$$\delta \quad ::= \quad nil \mid d :: \delta \qquad\qquad \textit{domain stacks}$$

The length *of a domain stack* $(d_1 :: \cdots :: d_n :: nil)$ *is* n*. The domain stack reversal function* rev *is defined as:*

$$\mathrm{rev}(d_1 :: \cdots :: d_n :: nil) \triangleq (d_n :: \cdots :: d_1 :: nil)$$

The notation $f[x \mapsto v]$ denotes the function which maps x to v and otherwise is equivalent to f. If $x \notin \mathrm{dom}(f)$, $f[x \mapsto v]$ denotes the function which extends f, mapping x to v. We define $\varphi(d, d') \triangleq \varphi(d) \cup \varphi(d')$. Substitution is defined as one may expect, with the following caveat:

Definition 5.2 *The identifier* $\mathbf{s}$ *is bound by the scope of objects, so in particular* $o[[\varrho]/\mathbf{s}] = o$*; otherwise, substitution is defined as usual.*

We define *frame depth* and *unframed* expressions in the same manner as Chapter 1, and similarly disallow framed subexpressions in objects of any sort. We then define well-formedness of configurations as follows:

Definition 5.3 *A configuration* $d :: \delta, e, \sigma$ *is* well-formed *iff* e *is closed and there exists* E *and unframed* e' *such that* $e = E[e']$ *and the frame depth of* E *equals the length of* δ*.*

Corollary 5.1 *If* $d :: \delta, e, \sigma$ *is well-formed and* $e = E[e']$ *with* e' *unframed, then the frame depth of* E *equals the length of* δ*.*

Hereafter we consider only well-formed configurations. It is easy to see that these well-formedness requirements are sensible and not overly restrictive via the following lemma, the proof of which follows by a straightforward (and tedious) case analysis, which is left as an exercise for the masochistic reader:

Lemma 5.1 *If a well-formed configuration* $d :: \delta, e, \sigma$ *is stuck, then* $e = E[e']$ *where* e' *is of the following form:*

1. $(co \cdot \varphi\backslash_\iota).m(v)$ *where* $m \in \iota$ *or* $m \notin \varphi(d, \partial)$
2. $[\varrho].m(v)$ *and* $(m(x) = e) \notin \varrho$

3. $(l \cdot \varphi\backslash_\iota).m(v)$ *and* $m \notin \{\text{set}, \text{get}\}$

4. $(co \cdot \varphi\backslash_\iota) \shortmid (d', \iota')$ *where* $\iota' \nsubseteq \varphi(d', \partial)$

5. $(l \cdot \varphi\backslash_\iota).\text{get}()$ *where* $l \notin \text{dom}(\sigma)$

6. $(l \cdot \varphi\backslash_\iota).\text{set}(v)$ *where* $l \notin \text{dom}(\sigma)$

The reflexive, transitive closure of $\rightarrow$ is denoted $\rightarrow^\star$. Other language relevant to properties of evaluation is defined as follows:

Definition 5.4 *The domain* d_1 *is the* top-level domain*. An expression* e *is* top-level *if it contains no subexpressions of the form* $\cdot e' \cdot$ *or* $[\varrho].m(e')$ *or* $l \cdot \varphi\backslash_\iota$*. If* $d_1 :: nil, e, \varnothing \rightarrow^\star d_1 :: nil, v, \sigma$ *for top-level* e*, we say that* e evaluates to v*. If there does not exist* v *such that* e *evaluates to* v *then* e diverges*, and if* $d_1 :: nil, e, \varnothing \rightarrow^\star d_1 :: nil, e', \sigma$ *and* $d_1 :: nil, e', \sigma$ *is stuck, then* e goes wrong.

An important feature of the semantics is that it requires that every domain has *at least* the default access rights to an object. In the *send* rule, we always require a test to ensure that the active protection domain is authorized for the specified use of the object: this detail is the essence of our *use*-based security model, as opposed to *communication*-based, in the sense that authorization for object access is checked when the object is used, not when it is communicated via message send or assignment. The **weak** mechanism semantics ensures that any return value from a message send to a weakened object is similarly weakened, and that the message send itself is allowable with respect to the weakening. The *cast* rule requires that any cast *restricts* access rights to a capability, so that increasing rights beyond the initial policy specification is disallowed. we will see in Sect. 5.4, the pop type system statically enforces all of these checks, so that the authorization checks associated with casting, weakening, and message sends, may be safely removed from the runtime system.

The self variable and self objects

In order for objects to always have complete access to themselves, the semantics specifies a rule for the use of self objects which imposes no run-time authorization checks; indeed, self objects have no interface or weakenings. The restriction that the variable **s** cannot appear unselected—that is, if **s** occurs in a program it must always be in an expression of the form $\mathbf{s}.m(e)$—ensures that **s** cannot escape its own scope. This implies that giving **s** "full strength" is safe, since it cannot provide a "back-door" to the object by being communicated outside. Rights amplification via self, discussed in Sect. 5.1, is still possible, but this is a *feature* of capability-based security, not a flaw of the model.

5.3 The pop-to-pml_B transformation

In this section we define the pop-to-pml_B transformation. We begin by defining a transformation of pop user interfaces into pml_B records with default values, denoted $\hat{\varphi}$, as follows:

$$\{d_1 \mapsto \iota_1, \ldots, \widehat{d_n \mapsto} \iota_n, \partial \mapsto \iota\} \quad = \quad \{\varnothing\}\{\partial = \iota\}\{d_1 = \iota_1\}\cdots\{d_n = \iota_n\}$$

In words, interface definitions are encoded as rows with fields indexed by domain names, including the default domain. Also, for brevity in the transformation definition we define the following syntactic sugar:

$$\begin{array}{rcll}
\{m_1 = e_1, \ldots, m_n = e_n\} & \triangleq & \{\varnothing\}\{m_1 = e_1\}\cdots\{m_n = e_n\} & \\
\mathsf{fix}\,\mathsf{s}.\lambda_.e & \triangleq & \mathsf{fix}\,\mathsf{s}.\lambda x.e & x \text{ not free in } e \\
e_1; e_2 & \triangleq & \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 & x \text{ not free in } e_2 \\
e \supseteq \iota & \triangleq & e \ni m_1; \ldots; e \ni m_n & \iota = \{m_1, \ldots, m_n\}
\end{array}$$

The pop-to-pml_B transformation is then defined in Fig. 5.3. The translation is effected by transforming pop objects into rows with obj fields containing method transformations, ifc fields containing interface transformations, and strong fields containing sets denoting methods on which the object is *not* weak.

Of technical interest is the use of pml_B lambda abstractions with recursive binding to encode the self variable $\mathbf{s}$ in the transformation. Also of technical note is the manner in which weakenings are encoded. In a pop weakened object $\mathbf{weak}_\iota(o)$, the set ι denotes the methods which are inaccessible via weakening. In the encoding these sets are turned "inside out", so that the strong field in objects denotes the fields which *are* accessible; in an unweakened object definition, this field contains $\bar{\varnothing}$. Accordingly, in the translation of message sends, any resulting composition of weakenings is encoded as an *intersection* of the composed strong fields, rather than a union. We define the translation in this manner to allow a simple definition of set subtyping, as well as typings of set operations in the pml_B type system, which translate into a simpler direct type system for pop. See Sect. 5.4 for details.

As for the λ_{sec}-to-pml_B transformation defined in Chapter 4, we will prove correctness of the pop-to-pml_B transformation, in the sense that the transformation preserves program semantics.

$$
\begin{array}{rcl}
[\![x]\!]_d & = & x \\
[\![\mathbf{s}.m(e)]\!]_d & = & (\mathbf{s}\{\}.m)[\![e]\!]_d \\
[\![[m_i(x) = e_i \;^{0<i\leq n}] \cdot d' \cdot \varphi\backslash_\iota]\!]_d & = & \{\mathsf{obj} = \mathsf{fix}\,\mathsf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_{d'} \;^{0<i\leq n}\}, \\
 & & \;\; \text{ifc} = \hat{\varphi}, \\
 & & \;\; \text{strong} = \bar{\iota}\} \\
[\![e_1.m(e_2)]\!]_d & = & \mathsf{let}\, os = \{o_1 = [\![e_1]\!]_d, o_2 = [\![e_2]\!]_d\}\,\mathsf{in} \\
 & & \mathsf{let}\, i = os.o_1.\text{ifc}\,\mathsf{in} \\
 & & \mathsf{let}\, w = os.o_1.\text{strong}\,\mathsf{in} \\
 & & \mathsf{let}\, o = (os.o_1.\mathsf{obj})\{\}\,\mathsf{in} \\
 & & ((i.d \vee i.\partial) \wedge w) \ni m; \\
 & & \mathsf{let}\, o_3 = o.m(os.o_2)\,\mathsf{in} \\
 & & o_3\{\text{strong} = (w \wedge o_3.\text{strong})\} \\
[\![e\,\mathord{\restriction}\,(d', \iota)]\!]_d & = & \mathsf{let}\, o_1 = [\![e]\!]_d\,\mathsf{in} \\
 & & (o_1.\text{ifc}.d' \vee o_1.\text{ifc}.\partial) \supseteq \iota; \\
 & & o_1\{\text{ifc} = ((o_1.\text{ifc})\{d' = \iota\})\} \\
[\![\mathbf{weak}_\iota(e)]\!]_d & = & \mathsf{let}\, o_1 = [\![e]\!]_d\,\mathsf{in} \\
 & & o_1\{\text{strong} = (\bar{\iota} \wedge o_1.\text{strong})\} \\
[\![\text{ref}_\varphi e]\!]_d & = & \mathsf{let}\, x = \text{ref}\; [\![e]\!]_d\,\mathsf{in} \\
 & & \mathsf{let}\, o = \lambda\_.\{\text{get} = \lambda y.!x, \text{set} = \lambda y.x := y\}\,\mathsf{in} \\
 & & \{\mathsf{obj} = o, \text{ifc} = \hat{\varphi}, \text{strong} = \bar{\varnothing}\} \\
[\![\mathsf{let}\, x = v\,\mathsf{in}\, e]\!]_d & = & \mathsf{let}\, x = [\![v]\!]_d\,\mathsf{in}\,[\![e]\!]_d
\end{array}
$$

Figure 5.3: The pop-to-pml$_B$ term transformation

It is a simulation result which, aside from providing confidence in the faithfulness of the transformation, will allow us to immediately obtain an indirect type soundness result for pop based on soundness of the pml_B type system, and will make direct type soundness for pop easier to prove as well. The desired property is stated as follows, and proved in the next section:

Theorem 5.1 **(pop-to-pml_B transformation correctness)** *If e evaluates to v then $[\![e]\!]_{d_1}$ evaluates to $[\![v]\!]_{d_1}$. If e diverges then so does $[\![e]\!]_{d_1}$. If e goes wrong then $[\![e]\!]_{d_1}$ goes wrong.*

5.3.1 Properties

Our proof of Theorem 5.1 will be based on an induction on arbitrary computations in pop. However, to state the induction properly, it is necessary to extend the pop-to-pml_B transformation to run-time entities, as follows:

Definition 5.5 *To treat* pop *run-time entities, we extend the transformation via the following definitions:*

1. $[\![[m_i(x) = e_i \ ^{0<i\leq n}].m(e)]\!]_d \ = \ ((\mathsf{fix}\,\mathsf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_d \ ^{0<i\leq n}\})\{\}.m)[\![e]\!]_d$
2. $[\![l \cdot \varphi \backslash_\iota]\!]_d \ = \ \{\mathrm{obj} = \lambda\_.\{\mathrm{get} = \lambda y.!l, \mathrm{set} = \lambda y.l := y\}, \mathrm{ifc} = \hat{\varphi}, \mathrm{strong} = \bar{\iota}\}$
3. $[\![\{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\}]\!] \ = \ \{l_1 \mapsto [\![v_1]\!]_d, \ldots, l_n \mapsto [\![v_n]\!]_d\}$

Note that in the above definition, the transformation of values in stores may be parameterized by arbitrary domain labels d; the following Lemma demonstrates that this is reasonable, since the transformation of values $[\![v]\!]_d$ does not depend on d:

Lemma 5.2 *For all d and d', if $[\![v]\!]_d$ is defined then $[\![v]\!]_d = [\![v]\!]_{d'}$.*

Proof. Immediate by definition of the transformation, since for any case of v the identifier d does not appear in the RHS of the definition of $[\![v]\!]_d$. □

Next, we define two *substitution* Lemmas relevant to the transformation:

Lemma 5.3 *If $[\![e]\!]_d$ is defined then $[\![e]\!]_d[[\![v]\!]_{d'}/x] = [\![e[v/x]]\!]_d$.*

Proof. By structural induction on e. In the basis we have $e = x'$, where by definition $[\![x]\!]_d = x'$. If $x' \neq x$ then $[\![e]\!]_d[[\![v]\!]_{d'}/x] = [\![e[v/x]]\!]_d = x'$. Otherwise we have $[\![x]\!]_d[[\![v]\!]_{d'}/x] = [\![v]\!]_{d'}$; but $[\![v]\!]_{d'} = [\![v]\!]_d$ by Lemma 5.2, and $[\![x[v/x]]\!]_d = [\![v]\!]_d$, so this case holds. The other cases follow in a straightforward manner by the induction hypothesis. □

Lemma 5.4 *Let* $s = [m_i(x) = e_i \ ^{0<i\leq n}]$ *and let* $v = \mathsf{fix}\, \mathsf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_d \ ^{0<i\leq n}\}$. *Then if* $[\![e]\!]_d$ *is defined,* $[\![e]\!]_d[v/\mathbf{s}] = [\![e[s/\mathbf{s}]]\!]_d$.

Proof. By structural induction on e. In the basis we have $e = x$ such that $x \neq \mathbf{s}$, since $[\![\mathbf{s}]\!]_d$ is undefined, so that $x[s/\mathbf{s}] = x$ and $[\![x]\!]_d = x$ by definition, therefore $[\![e]\!]_d[v/\mathbf{s}] = [\![e[s/\mathbf{s}]]\!]_d = x$. The induction step proceeds by case analysis on e. In case $e = \mathbf{s}.m(e')$, we have that $e[s/\mathbf{s}] = [m_i(x) = e_i \ ^{0<i\leq n}].m(e'[s/\mathbf{s}])$ by definition of self substitution, and:

$$[\![[m_i(x) = e_i \ ^{0<i\leq n}].m(e')]\!]_d = ((\mathsf{fix}\, \mathsf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_d \ ^{0<i\leq n}\})\{\}.m)[\![e'[s/\mathbf{s}]]\!]_d$$

by definition of the transformation. But $[\![\mathbf{s}.m(e')]\!]_d = (\mathbf{s}\{\}.m)[\![e']\!]_d$, so that:

$$((\mathbf{s}\{\}.m)[\![e']\!]_d)[v/\mathbf{s}] = ((\mathsf{fix}\, \mathsf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_d \ ^{0<i\leq n}\})\{\}.m)([\![e']\!]_d[v/\mathbf{s}])$$

and $[\![e'[s/\mathbf{s}]]\!]_d = [\![e']\!]_d[v/\mathbf{s}]$ by the induction hypothesis, so this case holds. The other cases follow in a straightforward manner by the induction hypothesis. $\square$

We may now prove the core of our simulation result, by showing that one-step reductions $\hookrightarrow$ may be simulated via the transformation.

Lemma 5.5 *The following assertions hold:*

1. *If* $d :: \delta, e_1, \sigma \hookrightarrow d' :: d :: \delta, \cdot e_2 \cdot, \sigma$ *by send, then* $[\![e_1]\!]_d, [\![\sigma]\!] \to^\star [\![e_2]\!]_{d'}, [\![\sigma]\!]$.
2. *If* $d' :: d :: \delta, \cdot v \cdot, \sigma \hookrightarrow d :: \delta, v, \sigma$ *by pop, then* $[\![v]\!]_d, [\![\sigma]\!] \to^\star [\![v]\!]_{d'}, [\![\sigma]\!]$.
3. *If* $d :: \delta, e_1, \sigma_1 \hookrightarrow d :: \delta, e_2, \sigma_2$ *by some rule besides send or pop, then* $[\![e_1]\!]_d, [\![\sigma_1]\!] \to^\star [\![e_2]\!]_d, [\![\sigma_2]\!]$.

Proof. Each assertion is treated individually:

1. In this case by definition of *send* we have:

$$\begin{aligned} e_1 &= ([\varrho] \cdot d' \cdot \varphi\backslash_\iota).m(v) \qquad \text{where } \varrho = (m_i(x) = e_i \ ^{0<i\leq n}) \text{ and } (m(x) = e) \in \varrho \\ e_2 &= \mathbf{weak}_\iota(e[[\varrho]/\mathbf{s}][v/x]) \end{aligned}$$

and $m \in (\varphi(d, \partial)\backslash\iota)$, therefore $m \notin \iota$ and $m \in \varphi(d, \partial)$. But by definition of the transformation we have:

$$\begin{aligned} [\![[\varrho] \cdot d' \cdot \varphi\backslash_\iota]\!]_d = \ & \{\text{obj} = \mathsf{fix}\, \mathsf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_{d'} \ ^{0<i\leq n}\}, \\ & \text{ifc} = \hat{\varphi}, \\ & \text{strong} = \bar{\iota}\} \end{aligned}$$

therefore, letting $v' = \mathsf{fix}\,\mathbf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_{d'}\ ^{0<i\leq n}\}$ we have:

$$
\begin{array}{rcl}
[\![[\varrho] \cdot d' \cdot \varphi\backslash_\iota]\!]_d.\text{ifc}, [\![\sigma]\!] & \rightarrow & \hat{\varphi}, [\![\sigma]\!] \\
[\![[\varrho] \cdot d' \cdot \varphi\backslash_\iota]\!]_d.\text{strong}, [\![\sigma]\!] & \rightarrow & \bar{\iota}, [\![\sigma]\!] \\
([\![[\varrho] \cdot d' \cdot \varphi\backslash_\iota]\!]_d.\text{obj})\{\}, [\![\sigma]\!] & \rightarrow^\star & \{m_i = \lambda x.[\![e_i]\!]_{d'}[v'/\mathbf{s}]\ ^{0<i\leq n}\}, [\![\sigma]\!]
\end{array}
$$

Further, since $m \notin \iota$ and $m \in \varphi(d, \partial)$, it is the case that $m \in (\hat{\varphi}.d \cup \hat{\varphi}.\partial)$ and $m \in \bar{\iota}$, i.e. $m \in ((\hat{\varphi}.d \cup \hat{\varphi}.\partial) \cap \bar{\iota})$. Therefore we have that $[\![e]\!]_d, [\![\sigma]\!] \rightarrow^\star e'', [\![\sigma]\!]$ in this case, where:

$$e'' = \ \mathsf{let}\, o_3 = [\![e]\!]_{d'}[v'/\mathbf{s}][[\![v]\!]_d/x]\,\mathsf{in}\,(o_3\{\text{strong} = (\bar{\iota} \wedge o_3.\text{strong})\})$$

by definition of the transformation and pml_B reduction. But by Lemma 5.4 and Lemma 5.3 we have that $([\![e]\!]_{d'}[v'/\mathbf{s}][[\![v]\!]_d/x]) = [\![e[[\varrho]/\mathbf{s}][v/x]]\!]_{d'}$, so:

$$e'' = \ \mathsf{let}\, o_3 = [\![e[[\varrho]/\mathbf{s}][v/x]]\!]_{d'}\,\mathsf{in}\,(o_3\{\text{strong} = (\bar{\iota} \wedge o_3.\text{strong})\})$$

and e'' is equivalent to $[\![\mathbf{weak}_\iota(e[[\varrho]/\mathbf{s}][v/x])]\!]_{d'}$, that is, to $[\![e_2]\!]_{d'}$, so the assertion holds.

2. This assertion holds immediately by Lemma 5.2 and reflexivity of $\rightarrow$.

3. This assertion follows by case analysis on the remaining reduction rules.

Case *self*. In this case $\sigma_1 = \sigma_2$, $e_1 = [\varrho].m(v)$, where $\varrho = (m_i(x) = e_i\ ^{0<i\leq n})$ and $(m(x) = e) \in \varrho$, and $e_2 = e[[\varrho]/\mathbf{s}][v/x]$. Let $v' = \mathsf{fix}\,\mathbf{s}.\lambda\_.\{m_i = \lambda x.[\![e]\!]_d\ ^{0<i\leq n}\}$; then by definition of the transformation we have:

$$[\![e_1]\!]_d = (v'\{\}.m)[\![v]\!]_d$$

therefore:

$$[\![e_1]\!]_d, [\![\sigma_1]\!] \rightarrow^\star [\![e]\!]_d[v'/\mathbf{s}][[\![v]\!]_d/x], [\![\sigma_2]\!]$$

But by Lemma 5.3 and Lemma 5.4 we have that $[\![e]\!]_d[v'/\mathbf{s}][[\![v]\!]_d/x] = [\![e[[\varrho]/\mathbf{s}][v/x]]\!]_d = [\![e_2]\!]_d$, so this case holds.

Case *cast*. In this case $\sigma_1 = \sigma_2$ and $e_1 = (co \cdot \varphi\backslash_\iota) \upharpoonright (d', \iota')$ and $e_2 = (co \cdot (\varphi[d' \mapsto \iota'])\backslash_\iota)$, with $\iota' \subseteq \varphi(d', \partial)$. Let $e = (co \cdot \varphi\backslash_\iota)$; then there exists e' such that:

$$
\begin{array}{rcl}
[\![e]\!]_d & = & \{\text{obj} = e', \text{ifc} = \hat{\varphi}, \text{strong} = \bar{\iota}\} \\
[\![e_2]\!]_d & = & \{\text{obj} = e', \text{ifc} = \widehat{\varphi[d' \mapsto \iota']}, \text{strong} = \bar{\iota}\}
\end{array}
$$

by definition of the transformation. Further:

$$
\begin{array}{rl}
[\![e_1]\!]_d = & \mathsf{let}\, o_1 = [\![e]\!]_d\,\mathsf{in} \\
& (o_1.\text{ifc}.d' \vee o_1.\text{ifc}.\partial) \supseteq \iota'; \\
& o_1\{\text{ifc} = ((o_1.\text{ifc})\{d' = \iota'\})\}
\end{array}
$$

Let $o_1 = [\![e]\!]_d$. Since $\iota' \subseteq \varphi(d', \partial)$, therefore $\iota \subseteq (o_1.\text{ifc}.d' \cup o_1.\text{ifc}.\partial)$, and since $o_1.\text{ifc}, [\![\sigma_2]\!] \rightarrow^\star \hat{\varphi}, [\![\sigma_2]\!]$ we have:

$$(o_1.\text{ifc})\{d' = \iota'\}, [\![\sigma_2]\!] \rightarrow^\star (\widehat{\varphi[d' \mapsto \iota']}), [\![\sigma_2]\!]$$

etc., therefore $[\![e_1]\!]_d, [\![\sigma_1]\!] \rightarrow^\star [\![e_2]\!]_d, [\![\sigma_2]\!]$ in this case by definition of $\rightarrow^\star$.

Case *weaken*. In this case $\sigma_1 = \sigma_2$, $e_1 = \mathbf{weak}_\iota(co \cdot \varphi\backslash_{\iota'})$ and $e_2 = co \cdot \varphi\backslash_{(\iota \cup \iota')}$. Let $e = co \cdot \varphi\backslash_{\iota'}$; then by definition of the transformation there exists e' such that:

$$\begin{aligned} [\![e]\!]_d &= \{\text{obj} = e', \text{ifc} = \hat{\varphi}, \text{strong} = \bar{\iota}\} \\ [\![e_2]\!]_d &= \{\text{obj} = e', \text{ifc} = \widehat{\varphi}, \text{strong} = \bar{\iota'} \cap \bar{\iota}\} \end{aligned}$$

and:

$$[\![e_1]\!]_d = \mathsf{let}\, o_1 = [\![e]\!]_d \,\mathsf{in}\, o_1\{\text{strong} = (\bar{\iota'} \wedge o_1.\text{strong})\}$$

so clearly $[\![e_1]\!]_d, [\![\sigma_1]\!] \rightarrow [\![e_2]\!]_d, [\![\sigma_2]\!]$ in this case.

Case *newcell*. In this case $e_1 = \text{ref}_\varphi v$, $e_2 = l \cdot \varphi\backslash_\varnothing$, and $\sigma_2 = \sigma_1[l \mapsto v]$ where $l \notin \text{dom}(\sigma_1)$. By definition of the transformation we have:

$$\begin{aligned} [\![e_1]\!]_d &= \mathsf{let}\, x = \text{ref}\ [\![v]\!]_d \,\mathsf{in} \\ &\quad\ \mathsf{let}\, o = \lambda\_.\{\text{get} = \lambda y.!x, \text{set} = \lambda y.x := y\} \,\mathsf{in} \\ &\quad\ \{\text{obj} = o, \text{ifc} = \hat{\varphi}, \text{strong} = \bar{\varnothing}\} \\ [\![e_2]\!]_d &= \{\text{obj} = \lambda\_.\{\text{get} = \lambda y.!l, \text{set} = \lambda y.l := y\}, \text{ifc} = \hat{\varphi}, \text{strong} = \bar{\varnothing}\} \end{aligned}$$

Now, since $l \notin \text{dom}(\sigma)$, therefore $l \notin \text{dom}([\![\sigma]\!])$, so by definition of $\text{pml}_B \rightarrow^\star$:

$$[\![e_1]\!]_d, [\![\sigma_1]\!] \rightarrow^\star [\![e_2]\!]_d, [\![\sigma_1]\!][l \mapsto [\![v]\!]_d]$$

But $[\![\sigma_1]\!][l \mapsto [\![v]\!]_d] = [\![\sigma_1[l \mapsto v]]\!]$ by definition and Lemma 5.2, so this case holds.

Case *set*. In this case $e_1 = (l \cdot \varphi\backslash_\iota).\text{set}(v)$ where $l \in \text{dom}(\sigma)$ and $\text{set} \in (\varphi(d, \partial)\backslash\iota)$, $e_2 = \mathbf{weak}_\iota(v)$ and $\sigma_2 = \sigma_1[l \mapsto v]$. Then by definition of the transformation we have:

$$[\![l \cdot \varphi\backslash_\iota]\!]_d = \{\text{obj} = \lambda\_.\{\text{get} = \lambda y.!l, \text{set} = \lambda y.l := y\}, \text{ifc} = \hat{\varphi}, \text{strong} = \bar{\iota}\}$$

so by definition of $\rightarrow^\star$ we have:

$$\begin{aligned} [\![l \cdot \varphi\backslash_\iota]\!]_d.\text{ifc}, [\![\sigma]\!] &\rightarrow \hat{\varphi}, [\![\sigma]\!] \\ [\![l \cdot \varphi\backslash_\iota]\!]_d.\text{strong}, [\![\sigma]\!] &\rightarrow \bar{\iota}, [\![\sigma]\!] \\ ([\![l \cdot \varphi\backslash_\iota]\!]_d.\text{obj})\{\}, [\![\sigma]\!] &\rightarrow^\star \{\text{get} = \lambda y.!l, \text{set} = \lambda y.l := y\}, [\![\sigma]\!] \end{aligned}$$

Further, since set $\in (\varphi(d, \partial)\backslash\iota)$ therefore set $\notin \iota$ and set $\in \varphi(d, \partial)$, so it is the case that set $\in (\hat{\varphi}.d \cup \hat{\varphi}.\partial)$ and set $\in \bar{\iota}$, i.e. set $\in ((\hat{\varphi}.d \cup \hat{\varphi}.\partial) \cap \bar{\iota})$. Therefore we have that:

$$[\![e_1]\!]_d, [\![\sigma_1]\!] \to^\star e'', [\![\sigma_1]\!][l \mapsto [\![v]\!]_d]$$

in this case, where:

$$e'' = \ \mathsf{let}\, o_3 = [\![v]\!]_d \,\mathsf{in}\, (o_3\{\text{strong} = (\bar{\iota} \wedge o_3.\text{strong})\})$$

by definition of the transformation and pml_B reduction. But e'' is equivalent to $[\![e_2]\!]_d$, and $[\![\sigma_1]\!][l \mapsto [\![v]\!]_d] = [\![\sigma_1[l \mapsto v]]\!]_d$ by definition, so this case holds.

Case *get*. In this case $e_1 = (l \cdot \varphi\backslash_\iota).\text{get}()$ where get $\in (\varphi(d, \partial)\backslash\iota)$, $\sigma_1 = \sigma_2$ and $e_2 = \mathbf{weak}_\iota(\sigma_2(l))$. Then by definition of the transformation we have:

$$[\![l \cdot \varphi\backslash_\iota]\!]_d \quad = \quad \{\text{obj} = \lambda\_.\{\text{get} = \lambda y.!l, \text{set} = \lambda y.l := y\}, \text{ifc} = \hat{\varphi}, \text{strong} = \bar{\iota}\}$$

so by definition of $\to^\star$ we have:

$$\begin{array}{rcl} [\![l \cdot \varphi\backslash_\iota]\!]_d.\text{ifc}, [\![\sigma]\!] & \to & \hat{\varphi}, [\![\sigma]\!] \\ [\![l \cdot \varphi\backslash_\iota]\!]_d.\text{strong}, [\![\sigma]\!] & \to & \bar{\iota}, [\![\sigma]\!] \\ ([\![l \cdot \varphi\backslash_\iota]\!]_d.\text{obj})\{\}, [\![\sigma]\!] & \to^\star & \{\text{get} = \lambda y.!l, \text{set} = \lambda y.l := y\}, [\![\sigma]\!] \end{array}$$

Further, since get $\in (\varphi(d, \partial)\backslash\iota)$ therefore get $\notin \iota$ and get $\in \varphi(d, \partial)$, so it is the case that get $\in (\hat{\varphi}.d \cup \hat{\varphi}.\partial)$ and get $\in \bar{\iota}$, i.e. get $\in ((\hat{\varphi}.d \cup \hat{\varphi}.\partial) \cap \bar{\iota})$. Therefore we have that $[\![e_1]\!]_d, [\![\sigma_1]\!] \to^\star e'', [\![\sigma_2]\!]$ in this case, where:

$$e'' = \ \mathsf{let}\, o_3 = [\![\sigma_2]\!](l) \,\mathsf{in}\, (o_3\{\text{strong} = (\bar{\iota} \wedge o_3.\text{strong})\})$$

by definition of the transformation and pml_B reduction. Let $v = \sigma_2(l)$; then $[\![\sigma_2]\!](l) = [\![v]\!]_d$ by definition and Lemma 5.2, so e'' is equivalent to $[\![e_2]\!]_d$ by definition, therefore this case holds.

Case *let* follows trivially by Lemma 5.3. $\square$

Before turning to arbitrary-length computations, we state another result relevant to the expression transformation:

Lemma 5.6 *For all v, if v is a closed value and $[\![v]\!]_d$ is defined, then $[\![v]\!]_d$ is a value.*

$$
\begin{array}{rcl}
\llbracket [\,] \rrbracket_{\delta} & = & [\,] \\
\llbracket [m_i(x) = e_i \ ^{0<i\leq n}].m(E) \rrbracket_{d::\delta} & = & ((\mathsf{fix\,s}.\lambda\_.\{m_i = \lambda x.\llbracket e_i \rrbracket_d \ ^{0<i\leq n}\})\{\}.m)\llbracket E \rrbracket_{d::\delta} \\
\llbracket E.m(e) \rrbracket_{d::\delta} & = & \mathsf{let}\, os = \{o_1 = \llbracket E \rrbracket_{d::\delta}, o_2 = \llbracket e \rrbracket_d\} \,\mathsf{in} \\
& & \mathsf{let}\, i = os.o_1.\mathrm{ifc} \,\mathsf{in} \\
& & \mathsf{let}\, w = os.o_1.\mathrm{strong} \,\mathsf{in} \\
& & \mathsf{let}\, o = (os.o_1.\mathrm{obj})\{\} \,\mathsf{in} \\
& & ((i.d \vee i.\partial) \wedge w) \ni m; \\
& & \mathsf{let}\, o_3 = o.m(os.o_2) \,\mathsf{in} \\
& & o_3\{\mathrm{strong} = (w \wedge o_3.\mathrm{strong})\} \\
\llbracket v.m(E) \rrbracket_{d::\delta} & = & \mathsf{let}\, os = \{o_1 = \llbracket v \rrbracket_d, o_2 = \llbracket E \rrbracket_{d::\delta}\} \,\mathsf{in} \\
& & \mathsf{let}\, i = os.o_1.\mathrm{ifc} \,\mathsf{in} \\
& & \mathsf{let}\, w = os.o_1.\mathrm{strong} \,\mathsf{in} \\
& & \mathsf{let}\, o = (os.o_1.\mathrm{obj})\{\} \,\mathsf{in} \\
& & ((i.d \vee i.\partial) \wedge w) \ni m; \\
& & \mathsf{let}\, o_3 = o.m(os.o_2) \,\mathsf{in} \\
& & o_3\{\mathrm{strong} = (w \wedge o_3.\mathrm{strong})\} \\
\llbracket e \restriction (d, \iota) \rrbracket_{\delta} & = & \mathsf{let}\, o_1 = \llbracket E \rrbracket_{\delta} \,\mathsf{in} \\
& & (o_1.\mathrm{ifc}.d \vee o_1.\mathrm{ifc}.\partial) \supseteq \iota; \\
& & o_1\{\mathrm{ifc} = ((o_1.\mathrm{ifc})\{d = \iota\})\} \\
\llbracket \mathbf{weak}_{\iota}(E) \rrbracket_{\delta} & = & \mathsf{let}\, o_1 = \llbracket E \rrbracket_{\delta} \,\mathsf{in} \\
& & o_1\{\mathrm{strong} = (\bar{\iota} \wedge o_1.\mathrm{strong})\} \\
\llbracket \mathrm{ref}_{\varphi} E \rrbracket_{\delta} & = & \mathsf{let}\, x = \mathrm{ref}\ \llbracket E \rrbracket_{\delta} \,\mathsf{in} \\
& & \mathsf{let}\, o = \lambda\_.\{\mathrm{get} = \lambda y.!x, \mathrm{set} = \lambda y.x := y\} \,\mathsf{in} \\
& & \{\mathrm{obj} = o, \mathrm{ifc} = \hat{\varphi}, \mathrm{strong} = \bar{\varnothing}\} \\
\llbracket \cdot E \cdot \rrbracket_{d::\delta} & = & \llbracket E \rrbracket_{\delta}
\end{array}
$$

Figure 5.4: The $\mathrm{pop\text{-}to\text{-}pml}_B$ evaluation context transformation

Proof. Immediate by definition of the transformation; the only closed value form for which v is undefined is $v = [\varrho]$; but since unselected self is disallowed in programs, this is a reasonable situation. □

Now, we consider arbitrary-length computations with respect to $\rightarrow^\star$. To perform the necessary analysis, we extend the pop-to-pml_B transformation to evaluation contexts in a straightforward manner. The context transformation is defined in Fig. 5.4. Similar to the the $\lambda^{\mathsf{S}}_{\text{sec}}$-to-$\lambda_{\text{sec}}$ simulation in Chapter 4, we will apply transformations to contexts along with the *reverse* of domain stacks in a configuration, since the oldest stack frames will apply to the outermost variables in contexts. We note that the current transformation is a indeed a transformation from contexts to contexts:

Lemma 5.7 *For all closed E, any defined $[\![E]\!]_\delta$ is a well-formed evaluation context.*

Proof. Immediate by definition of the context transformation; the only mildly interesting case is $E = v.m(E)$, but in this case $[\![E]\!]_\delta$ is well-formed by Lemma 5.6. □

Then, we prove some relevant properties of the transformation:

Lemma 5.8 *The following properties hold:*

1. *If $[\![E[e]]\!]_d$ is defined, then the frame depth of E is 0*
2. *If $[\![E[e]]\!]_d$ is defined, then $[\![E[e]]\!]_d = [\![E]\!]_{d::nil}[[\![e]\!]_d]$*
3. *If $E = E_1[E_2]$ where the frame depth of E_1 equals the length of δ, and $[\![E]\!]_{\text{rev}(d::\delta)}$ is defined, then $[\![E]\!]_{\text{rev}(d::\delta)} = [\![E_1]\!]_{\text{rev}(d::\delta)}[[\![E_2]\!]_{d::nil}]$*

Proof. Each assertion is treated individually:

1. Immediate by definition of $[\![E[e]]\!]_d$, since the transformation is defined only on unframed expressions.

2. By structural induction on E. In the basis $E = [\,]$, and since $[\![[\,]]\!]_{d::nil} = [\,]$, therefore $[\![E[e]]\!]_d = [\![E]\!]_{d::nil}[[\![e]\!]_d] = [\![e]\!]_d$ in this case. The proof then proceeds by case analysis on composite E, which excludes contexts of the form $\cdot E' \cdot$ by assertion 1.

Case $E = [m_i(x) = e_i \ ^{0<i\leq n}].m(E')$. In this case $[\![E]\!]_{d::nil}$ is equivalent to:

$$((\mathsf{fix}\,\mathsf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_d \ ^{0<i\leq n}\})\{\}.m)[\![E']\!]_{d::nil}$$

therefore $[\![E]\!]_{d::nil}[[\![e]\!]_d]$ is equivalent to:

$$((\mathsf{fix}\,\mathsf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_d \ ^{0<i\leq n}\})\{\}.m)[\![E']\!]_{d::nil}[[\![e]\!]_d]$$

and since $E[e] = [m_i(x) = e_i \ ^{0<i\leq n}].m(E'[e])$ in this case, $[\![E[e]]\!]_d$ is equivalent to:

$$((\mathsf{fix}\,\mathsf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_d \ ^{0<i\leq n}\})\{\}.m)[\![E'[e]]\!]_d$$

But by the induction hypothesis we have that $[\![E']\!]_{d::nil}[[\![e]\!]_d] = [\![E'[e]]\!]_d$, so the assertion holds in this case. The other cases follow in a similar, straightforward manner by the induction hypothesis, due to the tight correspondence of the term and context transformations.

3. By structural induction on E_1. In the basis we have that $E_1 = [\,]$, so $\delta = nil$. But $\mathrm{rev}(d :: nil) = (d :: nil)$, so that $[\![E]\!]_{\mathrm{rev}(d::\delta)} = [\![E_2]\!]_{d::nil}$, and $[\![[\,]]\!]_{\mathrm{rev}(d::nil)} = [\,]$, so that $[\![E_1]\!]_{\mathrm{rev}(d::\delta)}[[\![E_2]\!]_{d::nil}] = [\![E_2]\!]_{d::nil}$, therefore the basis holds. The induction step proceeds by case analysis on composite E_1.

Case $E_1 = \cdot E' \cdot$. Since the frame depth of E_1 equals the length of δ by assumption, therefore the length of δ is at least 1 in this case, hence $\delta = (d_1 :: \cdots :: d_n :: nil)$ for $n \geq 1$. Also, we have that $[\![E_1]\!]_{\mathrm{rev}(d::d_1::\cdots::d_n::nil)} = [\![E']\!]_{\mathrm{rev}(d::d_1::\cdots::d_{n-1}::nil)}$ and $[\![E_1[E_2]]\!]_{\mathrm{rev}(d::d_1::\cdots::d_n::nil)} = [\![E'[E_2]]\!]_{\mathrm{rev}(d::d_1::\cdots::d_{n-1}::nil)}$ by definition. But since the frame depth of E_1 equals the length of δ, therefore the frame depth of E' equals the length of $d_1 :: \cdots :: d_{n-1} :: nil$, so by the induction hypothesis we have that $[\![E']\!]_{\mathrm{rev}(d::d_1::\cdots::d_{n-1}::nil)}[[\![E_2]\!]_{d::nil}] = [\![E'[E_2]]\!]_{\mathrm{rev}(d::d_1::\cdots::d_{n-1}::nil)}$, therefore this case holds.

The other cases follow in a similar manner by the induction hypothesis. $\square$

Next, we define a simulation relation between pop and pml_B, in terms of the expression and context transformations:

Definition 5.6 *For all $d :: \delta$, pop expressions e and pml_B expressions e', the relation $(d :: \delta, e) \lhd e'$ holds iff there exists E_1 and e_1 such that $e = E_1[e_1]$, the frame depth of E_1 equals the length of δ, and $e' = [\![E_1]\!]_{\mathrm{rev}(d::\delta)}[[\![e_1]\!]_d]$.*

We then prove that this relation is a mapping:

Lemma 5.9 *If $(\delta, e) \lhd e'$ and $(\delta, e) \lhd e''$ then $e' = e''$.*

Proof. Let $\delta = (d :: \delta')$, and let E_1, E_1', e_1 and e_1' be such that $E_1[e_1] = E_1'[e_1'] = e$, with the frame depths of E_1 and E_1' equal to the length of δ' and $[\![E_1]\!]_{\mathrm{rev}(d::\delta')}[[\![e_1]\!]_d] = e'$ and $[\![E_1']\!]_{\mathrm{rev}(d::\delta')}[[\![e_1']\!]_d] = e''$. Assume w.l.o.g. that $e_1 = E[e_1']$ for some E, so that $E_1' = E_1[E]$. Since $[\![e_1]\!]_d$ is defined, therefore the frame depth of E is 0 by Lemma 5.8, so also by Lemma 5.8 we have that $[\![e_1]\!]_d = [\![E]\!]_{d::nil}[[\![e_1']\!]_d]$. Further, since the frame depth of E_1 equals the length of δ', therefore $[\![E_1']\!]_{\mathrm{rev}(d::\delta')} = [\![E_1]\!]_{\mathrm{rev}(d::\delta')}[[\![E]\!]_{d::nil}]$ by Lemma 5.8. But then $[\![E_1']\!]_{\mathrm{rev}(d::\delta')}[[\![e_1']\!]_d] = [\![E_1]\!]_{\mathrm{rev}(d::\delta')}[[\![E]\!]_{d::nil}[[\![e_1]\!]_d]] = [\![E_1]\!]_{\mathrm{rev}(d::\delta')}[[\![e_1]\!]_d]$, therefore $e' = e''$. □

The following result shows that the simulation relation may be preserved through one step of pop reduction:

Lemma 5.10 *If* $\delta_1, e_1, \sigma_1 \;\rightarrow\; \delta_2, e_2, \sigma_2$ *and* $(\delta_1, e_1) \lhd e_1'$ *then* $e_1', [\![\sigma_1]\!] \;\rightarrow^\star\; e_2', [\![\sigma_2]\!]$ *such that* $(\delta_2, e_2) \lhd e_2'$.

Proof. By *context* we have that $e_1 = E[e]$ and $e_2 = E[e']$ with $\delta_1, e, \sigma_1 \hookrightarrow \delta_2, e', \sigma_2$. The proof then proceeds by cases corresponding to those treated in the assertions enumerated in Lemma 5.5:

Case 1. In this case $\delta_1 = (d :: \delta)$, $\delta_2 = (d' :: d :: \delta)$, $\sigma_1 = \sigma_2$ and e' is of the form $\cdot e'' \cdot$ with $[\![e]\!]_d, [\![\sigma_1]\!] \rightarrow^\star [\![e'']\!]_{d'}, [\![\sigma_2]\!]$. Let $e_1' = [\![E]\!]_{\mathrm{rev}(\delta_1)}[[\![e]\!]_d]$ and $e_2' = [\![E[\cdot[\,]\cdot]]\!]_{\mathrm{rev}(\delta_2)}[[\![e'']\!]_{d'}]$. The frame depth of E equals the length of δ by Corollary 5.1, so also the frame depth of $E[\cdot[\,]\cdot]$ equals the length of $d :: \delta$, therefore we have that $(\delta_1, e_1) \lhd e_1'$ and $(\delta_2, e_2) \lhd e_2'$ by definition. But clearly $[\![E[\cdot[\,]\cdot]]\!]_{\mathrm{rev}(\delta_2)} = [\![E]\!]_{\mathrm{rev}(\delta_1)}$, so $e_1', [\![\sigma_1]\!] \rightarrow^\star e_2', [\![\sigma_2]\!]$ in this case by multiple applications of *context*.

Case 2. In this case $\delta_1 = (d' :: d :: \delta)$, $\delta_2 = (d :: \delta)$, $\sigma_1 = \sigma_2$, and e is of the form $\cdot v \cdot$ and $e' = v$. Let $e_1' = [\![E[\cdot[\,]\cdot]]\!]_{\mathrm{rev}(\delta_1)}[[\![v]\!]_{d'}]$ and $e_2' = [\![E]\!]_{\mathrm{rev}(\delta_2)}[[\![v]\!]_d]$. The frame depth of $E[\cdot[\,]\cdot]$ equals the length of δ_2 by well-formedness of configurations, so also the frame depth of E equals the length of δ, therefore we have that $(\delta_1, e_1) \lhd e_1'$, and $(\delta_2, e_2) \lhd e_2'$ by definition. But clearly $[\![E[\cdot[\,]\cdot]]\!]_{\mathrm{rev}(\delta_1)} = [\![E]\!]_{\mathrm{rev}(\delta_2)}$, and $[\![v]\!]_{d'} = [\![v]\!]_d$ by Lemma 5.2, hence $e_1' = e_2'$, so $e_1', [\![\sigma_1]\!] \rightarrow^\star e_2', [\![\sigma_2]\!]$ in this case by reflexivity of $\rightarrow^\star$.

Case 3. In this case $\delta_1 = \delta_2 = d :: \delta$, with $[\![e]\!]_d, [\![\sigma_1]\!] \rightarrow^\star [\![e']\!]_d, [\![\sigma_2]\!]$. Let $e_1' = [\![E]\!]_{\mathrm{rev}(\delta_1)}[[\![e]\!]_d]$ and $e_2' = [\![E]\!]_{\mathrm{rev}(\delta_2)}[[\![e']\!]_d]$. By definition of $\hookrightarrow$ both e and e' are unframed, so the frame depth of E in this case is equal to the length of δ by Corollary 5.1, hence $(\delta_1, e_1) \lhd e_1'$, and $(\delta_2, e_2) \lhd e_2'$ by definition. Furthermore, since $\delta_1 = \delta_2$ we have that $[\![E]\!]_{\mathrm{rev}(\delta_1)} = [\![E]\!]_{\mathrm{rev}(\delta_2)}$, so $e_1', [\![\sigma_1]\!] \rightarrow^\star e_2', [\![\sigma_2]\!]$ in this case by multiple applications of *context*. □

The previous result then generalizes easily to arbitrary computations, since the simulation relation is a mapping:

Lemma 5.11 *If* $\delta_1, e_1, \sigma_1 \rightarrow^\star \delta_2, e_2, \sigma_2$ *and* $(\delta_1, e_1) \lhd e_1'$ *then* $e_1', [\![\sigma_1]\!] \rightarrow^\star e_2', [\![\sigma_2]\!]$ *where* $(\delta_2, e_2) \lhd e_2'$.

Proof. Straightforward by Lemma 5.10 and induction on the length of the reduction $\delta_1, e_1, \sigma_1 \rightarrow^\star \delta_2, e_2, \sigma_2$. □

One final step before proving the main result is the observation that relevant dynamic properties of configurations are preserved in transformation:

Lemma 5.12 *If δ, e, σ is stuck and $(\delta, e) \lhd e'$, then $e', [\![\sigma]\!]$ goes wrong. If $(\delta, v) \lhd e'$ then e' is a value. If $(\delta, e) \lhd e'$ and e' is not a value nor of the form $\cdot v \cdot$, then e' is not a value.*

Proof. Suppose δ, e, σ is stuck; then $e = E[e']$ where e' is one of the forms specified in Lemma 5.1. For each form, it is easy to see that the transformation $[\![e']\!]_d, [\![\sigma]\!]$ will go wrong, and here we only sketch the relevant case analysis: if e is stuck because e' is a method select on m which is unauthorized to the active domain, or which has been disallowed by weakening, then the transformation implements a check which will also fail. If e is stuck because e' is a method select on m which does not exist in the object, then an m field will not exist in $[\![e']\!]_d$, so a projection of that field will fail. If e is stuck because e' is a set or a get on a cell object with location l such that $l \notin \sigma$, then $l \notin [\![\sigma]\!]$, so the transformation of these actions will also fail, as the transformation preserves store locations.

Suppose that $(\delta, v) \lhd e'$; then $e' = [\![v]\!]_d$ where $\delta = d :: \delta$ by definition of $\lhd$, and $[\![v]\!]_d$ is a value by Lemma 5.6.

Finally, suppose e is not a value nor of the form $\cdot v \cdot$. Let $\delta = d :: \delta$; since $(\delta, e) \lhd e'$, there exists E and e'' such that $e = E[e'']$ and $e' = [\![E]\!]_{\mathrm{rev}(\delta)}[\![e'']\!]_d$ by definition. Suppose that $E = [\,]$; then e'' is not a value, and clearly $[\![e'']\!]_d$ is not a value by definition of the transformation. Suppose E is composite; then clearly $[\![E]\!]_{\mathrm{rev}(\delta)}[\![e'']\!]_d$ is not a value by definition of the transformation, since E is not of the form $\cdot[\,]\cdot$ by assumption, and for any e''', $[\![E]\!]_{\mathrm{rev}(\delta)}[e''']$ is not a value in this case. $\square$

We may now restate and prove the principal result of this section, that is, the correctness of the pop-to-pml_B transformation, as follows:

Theorem 5.1 (pop**-to-**pml_B **transformation correctness)** *If e evaluates to v then $[\![e]\!]_{d_1}$ evaluates to $[\![v]\!]_d$. If e diverges then so does $[\![e]\!]_{d_1}$. If e goes wrong then $[\![e]\!]_{d_1}$ goes wrong.*

Proof. Suppose for top-level e we have $d_1 :: nil, e, \varnothing \to^\star d_1 :: nil, v, \sigma$. Then $(d_1 :: nil, e) \lhd [\![e]\!]_{d_1}$ and $(d_1 :: nil, v) \lhd [\![v]\!]_{d_1}$ by definition, and $[\![e]\!]_{d_1}, \varnothing \to^\star [\![v]\!]_{d_1}, [\![\sigma]\!]$ by Lemma 5.11 and Lemma 5.9, and $[\![v]\!]_{d_1}$ is a value by Lemma 5.12.

Suppose for top-level e we have that $d_1 :: nil, e, \varnothing$ does not terminate, and suppose on the contrary that there exists σ and v such that $[\![e]\!]_{d_1}, \varnothing \to^\star v, \sigma$. Since $(d_1 :: nil, e) \lhd [\![e]\!]_{d_1}$ by definition, by Lemma 5.11 there must exist e', σ' and δ such that $(\delta, e') \lhd v$ and $\sigma = [\![\sigma']\!]$ and $d_1 :: nil, e, \varnothing \to^\star \delta, e', \sigma'$, where e' is not a value nor of the form $\cdot v' \cdot$ by assumption, since in the latter case δ, e', σ' would evaluate to v' by definition of $\to^\star$ and well-formedness of configurations.

But then v is not a value by Lemma 5.12, which is a contradiction.

Finally, suppose for top-level e we have $d_1 :: nil, e, \varnothing \rightarrow^\star \delta, e', \sigma$ and δ, e', σ is stuck. Since $(d_1 :: nil, e) \lhd [\![e]\!]_{d_1}$ by definition, therefore $[\![e]\!]_{d_1}, \varnothing \rightarrow^\star e'', [\![\sigma]\!]$ such that $(\delta, e') \lhd e''$ by Lemma 5.11, and $e'', [\![\sigma]\!]$ goes wrong by Lemma 5.12. $\square$

5.4 Types for pop

In this section we introduce a let-polymorphic type analysis for pop, which we develop using the same transformational method described in Chapter 5, where type systems for λ_{sec} were defined. In particular, we easily obtain an *indirect* type analysis for pop via composition of the pop-to-pml_B transformation and pml_B type judgements, which is sound by Theorem 5.1 and Theorem 3.2. Furthermore, the development of, and soundness proof for, a *direct* pml_B type analysis is made significantly easier using this approach. This demonstrates the usefulness of the pml_B language, insofar as it may be used as a transformational target for two distinct source languages, λ_{sec} and pml_B.

5.4.1 Indirect types

The type systems $\mathcal{S}_i^{rel}$ for pml_B were specified in Definition 3.2. Sect. 5.3 defined a translation of pop into pml_B. Composing the two automatically gives rise to a type system for pop, whose safety is a direct consequence of Theorems 5.1 and 3.2.

Definition 5.7 *Let e be a closed* pop *expression. By definition, $C, \Gamma \vdash e : \sigma$ holds if and only if $C, \Gamma \vdash [\![e]\!]_{d_1} : \sigma$ holds.*

Theorem 5.2 (Indirect pop type soundness) *If e is a closed* pop *expression and $C, \Gamma \vdash [\![e]\!]_{d_1} : \sigma$ is valid, then e does not go wrong.*

As in the case of λ_{sec}, turning type safety into a trivial corollary was the principal motivation for basing our approach on a transformation. Indeed, because Theorem 5.1 concerns untyped terms, its proof is straightforward, and constitutes the principal proof effort for this pop soundness result.

$$\tau ::= \alpha, \beta, \ldots \mid \{\tau\} \mid [\tau] \mid [\tau]^{\tau}_{\tau} \mid m : \tau \to \tau \,;\, \tau \mid d : \tau \,;\, \tau \mid m, \tau \mid \epsilon$$

Figure 5.5: Direct pop type grammar

$$\frac{\alpha \in \mathcal{V}_k}{\alpha : k} \qquad \epsilon : \mathit{Meth}_M, \mathit{Set}_M, \mathit{Ifc}_D \qquad \frac{m \notin M \qquad \tau : \mathit{Set}_{M \cup \{m\}}}{m, \tau : \mathit{Set}_M}$$

$$\frac{\tau_1 : \mathit{Type} \qquad \tau_2 : \mathit{Type} \qquad m \notin M \qquad \tau : \mathit{Meth}_{M \cup \{m\}}}{m : \tau_1 \to \tau_2 \,;\, \tau : \mathit{Meth}_M} \qquad \frac{\tau : \mathit{Meth}_\varnothing}{[\tau] : \mathit{Type}}$$

$$\frac{\tau_1 : \mathit{Set}_\varnothing \qquad d \notin D \qquad \tau : \mathit{Ifc}_{D \cup \{d\}}}{d : \{\tau_1\} \,;\, \tau_2 : \mathit{Ifc}_D} \qquad \frac{\tau_1 : \mathit{Meth}_\varnothing \qquad \tau_2 : \mathit{Ifc}_\varnothing \qquad \tau_3 : \mathit{Set}_\varnothing}{[\tau_1]^{\{\tau_2\}}_{\{\tau_3\}} : \mathit{Type}}$$

Figure 5.6: Direct pop type kinding rules

5.4.2 Direct types

While this indirect type system is a sound static analysis for pop, it is desirable to define a direct static analysis for pop, for the same reasons that this was desirable in the case of λ_{sec}. That is, the term transformation required for the indirect analysis is an unwanted complication for compilation, the indirect type system is not a clear declaration of program properties for the programmer, and type error reporting would be extremely troublesome. Thus, we define a direct type system for pop, the development of which significantly benefits from the transformational approach. In particular, type soundness for the direct system may be demonstrated by a simple appeal to soundness in the indirect system, rather than *ab initio*.

While direct type system for pop is based on the pml_B type system, we also develop a specialized type language for the sake of readability, and for an intuitive correspondence with pop expressions. The direct type language for pop is defined in Fig. 5.5. The most novel feature of the pop type language is the form for objects $[\tau]^{\{\tau_1\}}_{\{\tau_2\}}$, where τ_2 is the type of any weakening set imposed on the object, and τ_1 is the type of its interface. Types of sets are essentially the sets themselves, modulo polymorphic features; we abbreviate a type of the form $\tau; \epsilon$ or τ, ϵ as τ. As always, we immediately restrict pop types to meaningful forms by requiring them to be well-kinded, with the

$$
\begin{array}{rcl}
(\!|\, [\tau] \,|\!) & = & \{\partial\{\varnothing\}\} \rightarrow \{(\!|\, \tau \,|\!)\} \\
(\!|\, [\tau_1]^{\{\tau_2\}}_{\{\tau_3\}} \,|\!) & = & \{\mathrm{obj} : (\!|\, [\tau_1] \,|\!)\ ;\ \mathrm{ifc} : \{(\!|\, \tau_2 \,|\!)\}\ ;\ \mathrm{strong} : \{(\!|\, \tau_3 \,|\!)_-\}\ ;\ \partial\{\varnothing\}\} \\
(\!|\, m : \tau_1 \rightarrow \tau_2\ ;\ \tau \,|\!) & = & m : (\!|\, \tau_1 \,|\!) \rightarrow (\!|\, \tau_2 \,|\!)\ ;\ (\!|\, \tau \,|\!) \\
(\!|\, d : \{\tau_1\}\ ;\ \tau_2 \,|\!) & = & d : \{(\!|\, \tau_1 \,|\!)_+\}\ ;\ (\!|\, \tau_2 \,|\!) \\
(\!|\, \epsilon \,|\!) & = & \partial\{\varnothing\} \\
(\!|\, \beta \,|\!), (\!|\, \beta \,|\!)_+, (\!|\, \beta \,|\!)_- & = & \beta \\
(\!|\, m, \tau \,|\!)_+ & = & m+, (\!|\, \tau \,|\!)_+ \\
(\!|\, \epsilon \,|\!)_+ & = & \varnothing \\
(\!|\, m, \tau \,|\!)_- & = & m-, (\!|\, \tau \,|\!)_- \\
(\!|\, \epsilon \,|\!)_- & = & \omega
\end{array}
$$

Figure 5.7: The pop-to-pml_B type transformation

relevant rules defined in Fig. 5.6.

The direct pop type language has a straightforward interpretation in the pml_B type language, defined in Fig. 5.7; since we will base the direct type system on $\mathcal{S}_1^{\leq}$, the model for the direct pop type language is thus the $\mathrm{RS}^{\leq_1}$ model defined in Sect. 3.2.2. The interpretation is extended to constraints and typing environments in the obvious manner. In this interpretation, we turn weakening sets "inside-out"; this is to allow the types of weakenings to correspond to disallowed method names, in keeping with the operational meaning of weakenings in pop. Turning these types inside-out in the type transformation also corresponds to the manner in which weakening sets are turned inside-out in the pop–to–pml_B language term transformation. One of the benefits of this approach is with regard to subtyping; weakening sets can be safely strengthened, and user interfaces safely weakened, in a uniform manner via subtyping coercions.

The direct type judgement system for pop, the rules for which are *derived* from $\mathcal{S}_1^{\leq}$ type judgements for transformed pop expressions, is defined in Fig. 5.8. For simplicity, we do not include constraints in type judgements, but rely only on atomic subtyping. The following definition describes new relations appearing in the pop type judgement rules:

Definition 5.8 *The relation* $\tau \leq \tau'$ *holds iff* **true** $\Vdash (\!|\, \tau \,|\!) \leq (\!|\, \tau' \,|\!)$ *holds in* $\mathrm{RS}^{\leq_1}$. *The relation*

$$\textsc{Interface}\quad \vdash \left\{d_i \mapsto \iota_i \ ^{0<i\leq n}\right\} : \{d_i : \{\iota_i\} \ ^{0<i\leq n}\}$$

$$\textsc{Var}\quad \frac{\Gamma(x) = \sigma}{d, \Gamma \vdash x : \sigma}$$

$$\textsc{Sub}\quad \frac{d, \Gamma \vdash e : \tau \qquad \tau \leq \tau'}{d, \Gamma \vdash e : \tau'}$$

$$\textsc{Let}\quad \frac{d, \Gamma \vdash v : \sigma \qquad d, (\Gamma; x : \sigma) \vdash e : \tau}{d, \Gamma \vdash \mathsf{let}\, x = v \,\mathsf{in}\, e : \tau}$$

$$\forall\ \textsc{Intro}\quad \frac{d, \Gamma \vdash e : \tau \qquad \bar{\alpha} \cap \mathrm{fv}(\Gamma) = \varnothing}{d, \Gamma \vdash e : \forall \bar{\alpha}.\tau}$$

$$\forall\ \textsc{Elim}\quad \frac{d, \Gamma \vdash e : \forall \bar{\alpha}.\tau'}{d, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau'}$$

$$\textsc{Ref}\quad \frac{d, \Gamma \vdash e : \tau \qquad \vdash \varphi : \tau_\varphi}{d, \Gamma \vdash \mathrm{ref}_\varphi e : [\mathrm{set} : \tau \to \tau, \mathrm{get} : \tau' \ \to \tau]^{\tau_\varphi}_{\{\epsilon\}}}$$

$$\textsc{Obj}\quad \frac{\vdash \varphi : \tau_\varphi \qquad \tau'_\varphi = d' : \{m_i \ ^{0<i\leq n}\} \qquad (d', \Gamma; x : \tau_j; \mathbf{s} : [m_i : \tau_i \to \tau'_i \ ^{0<i\leq n}] \vdash e_j : \tau'_j) \ ^{0<j\leq n}}{d, \Gamma \vdash [m_i(x) = e_i \ ^{0<i\leq n}] \cdot d' \cdot \varphi\backslash_\iota : [m_i : \tau_i \to \tau'_i \ ^{0<i\leq n}]^{\{\tau_\varphi\}}_{\{\iota\}}}$$

$$\textsc{Send}\quad \frac{d, \Gamma \vdash e_1 : [m : \tau' \to \tau^{\tau_\varphi}_{\{\tau_w\}} \ ; \ \tau_1]^{\{d:\{m,\ \tau_2\}\ ;\ \tau_3\}}_{\{\tau_w\}} \qquad d, \Gamma \vdash e_2 : \tau' \qquad m \notin \tau_w}{d, \Gamma \vdash e_1.m(e_2) : \tau^{\tau_\varphi}_{\{\tau_w\}}}$$

$$\textsc{Self}\quad \frac{d, \Gamma \vdash \mathbf{s} : [m : \tau' \to \tau \ ; \ \tau''] \qquad d, \Gamma \vdash e : \tau'}{d, \Gamma \vdash \mathbf{s}.m(e) : \tau}$$

$$\textsc{Cast}\quad \frac{d, \Gamma \vdash e : [\tau]^{\{d':\{m_i \ ^{0<i\leq n}\ ;\ \tau_1\}\ ;\ \tau_2\}}_{\tau_w} \qquad \iota = \left\{m_i \ ^{0<i\leq n}\right\}}{d, \Gamma \vdash e{\restriction}(d', \iota) : [\tau]^{\{d':\iota\ ;\ \tau_2\}}_{\tau_w}}$$

$$\textsc{Weak}\quad \frac{d, \Gamma \vdash e : \tau^{\tau_\varphi}_{\{m_i \ ^{0<i\leq n}, \tau'\}} \qquad \iota = \left\{m_i \ ^{0<i\leq n}\right\}}{d, \Gamma \vdash \mathbf{weak}_\iota(e) : \tau^{\tau_\varphi}_{\{m_i \ ^{0<i\leq n}, \tau'\}}}$$

Figure 5.8: Direct type judgements for pop

$m \notin \tau_w$ *holds iff* **true** $\not\Vdash \exists\beta.(\llparenthesis\, \tau_w \,\rrparenthesis_+ \leq \llparenthesis\, m, \beta \,\rrparenthesis_+)$ *holds in* $\mathrm{RS}^{\leq_1}$*, where* $\beta \notin \mathrm{fv}(C, \tau_w)$.

5.4.3 Direct type safety and optimizations

The easily proven, tight correlation between the indirect and direct pop type systems is clearly demonstrated with the following lemma, which follows in the same manner as Theorem 4.5; the proof is straightforward, since the direct type judgements can be viewed simply as syntactic sugar for $\mathcal{S}_{\mathrm{I}}^{\leq}$ judgements:

Lemma 5.13 $d, \Gamma \vdash e : \tau$ *is valid iff* **true**, $\llparenthesis\, \Gamma \,\rrparenthesis \vdash \llbracket e \rrbracket_d : \llparenthesis\, \tau \,\rrparenthesis$ *is in* $\mathcal{S}_{\mathrm{I}}^{\leq}$.

Then, along with Theorem 5.1, this correlation is sufficient to immediately establish direct type soundness for pop:

Theorem 5.3 (Direct type soundness) *If* e *is a closed* pop *expression and* $d, \Gamma \vdash e : \tau$ *is valid, then* e *does not go wrong.*

This result again demonstrates the advantages of the transformational method, which has allowed us to define a direct, expressive static analysis for pop with a minimum of proof effort.

The next lemma provides further confidence in our direct pop type system, by ensuring that the derived system is complete, in the sense that if a transformed pop expression has *any* type in the pml_B $\mathcal{S}_{\mathrm{I}}^{\leq}$ system (not necessarily a type which is the image of an pop–to–pml_B type transformation), then it will be typable in the derived system:

Lemma 5.14 *If* **true**, $\Gamma \vdash \llbracket e \rrbracket_d : \tau$ *is valid in* $\mathcal{S}_{\mathrm{I}}^{\leq}$*, then there exists* Γ', τ' *such that* **true**, $\llparenthesis\, \Gamma' \,\rrparenthesis \vdash \llbracket e \rrbracket_d : \llparenthesis\, \tau' \,\rrparenthesis$ *is valid.*

The proof follows in a straightforward manner by rule induction on the judgement $C, \Gamma \vdash \llbracket e \rrbracket_d : \tau$ and definition of the pop-to-pml_B transformation.

Another benefit of our static analysis for pop, as for pml_B and λ_{sec}, is that security checks in well-typed programs may be eliminated at run-time, since well-typed programs are guaranteed to be safe. The optimizations that may be effected for pop are particularly substantial; in fact, the only reason for user interfaces to have any run-time presence at all in well-typed programs is for the sake of separate compilation. The optimized semantics for pop is defined in Fig. 5.9. The safety of these semantics is verified with the following result, which follows by definition of $\rightarrow^\star$ and Theorem 5.3:

Corollary 5.2 *Let* e *be a closed* pop *expression; then* $d, e, \varnothing \rightsquigarrow^\star d', v, \sigma$ *iff* $d, e, \varnothing \rightarrow^\star d', v, \sigma$.

$$
\begin{array}{rcll}
d :: \delta, ([\varrho] \cdot d' \cdot \varphi\backslash_\iota).m(v), \sigma & \rightsquigarrow & d' :: d :: \delta, \cdot(\mathbf{weak}_\iota(e[[\varrho]/\mathbf{s}][v/x]))\cdot, \sigma & (\textit{send}) \\
 & & \qquad\qquad \text{if } (m(x) = e) \in \varrho & \\
\delta, [\varrho].m(v), \sigma & \rightsquigarrow & \delta, e[[\varrho]/\mathbf{s}][v/x], \sigma \qquad \text{if } (m(x) = e) \in \varrho & (\textit{self}) \\
\delta, (co \cdot \varphi\backslash_\iota) \upharpoonright (d', \iota'), \sigma & \rightsquigarrow & \delta, (co \cdot (\varphi[d' \mapsto \iota'])\backslash_\iota), \sigma & (\textit{cast}) \\
\delta, \mathbf{weak}_\iota(co \cdot \varphi\backslash_{\iota'}), \sigma & \rightsquigarrow & \delta, co \cdot \varphi\backslash_{(\iota \cup \iota')}, \sigma & (\textit{weaken}) \\
\delta, \mathrm{ref}_\varphi v, \sigma & \rightsquigarrow & \delta, l \cdot \varphi\backslash_\varnothing, \sigma[l \mapsto v] \qquad l \notin \mathrm{dom}(\sigma) & (\textit{newcell}) \\
d :: \delta, (l \cdot \varphi\backslash_\iota).\mathrm{set}(v), \sigma & \rightsquigarrow & d :: \delta, \mathbf{weak}_\iota(v), \sigma[l \mapsto v] \qquad l \in \mathrm{dom}(\sigma) & (\textit{set}) \\
d :: \delta, (l \cdot \varphi\backslash_\iota).\mathrm{get}(), \sigma & \rightsquigarrow & d :: \delta, \mathbf{weak}_\iota(\sigma(l)), \sigma & (\textit{get}) \\
\delta, \mathsf{let}\, x = v \,\mathsf{in}\, e, \sigma & \rightsquigarrow & \delta, e[v/x], \sigma & (\textit{let}) \\
d :: \delta, \cdot v \cdot, \sigma & \rightsquigarrow & \delta, v, \sigma & (\textit{pop}) \\
\delta, E[e], \sigma & \rightsquigarrow & \delta', E[e'], \sigma' \qquad \text{if } \delta, e, \sigma \rightsquigarrow \delta', e', \sigma' & (\textit{context})
\end{array}
$$

Figure 5.9: Optimized operational semantics for pop

Conceivably, the semantics can be optimized even further; since all access control with respect to weakening and interfaces is enforced statically, it is no longer necessary to propagate this information through run-time. However, we maintain the information in the optimized semantics for the purposes of modularity.

5.5 Examples and discussion

In this section we provide several examples that demonstrate the usage and flexibility of the pop system, including a scheme for embedding the ownership types of [5] in pop in a type-safe manner, as well as a scheme for encoding class definitions with `public`, `private` and `protected` instance modifiers.

5.5.1 Basic typing examples

Here is a brief example illustrating the features of pop and the expressiveness of its direct type system. We may create a cell c which is read-write in domain d but read-only elsewhere, containing a value v, as follows:

$$c = \mathrm{ref}_{\{d \mapsto \{\text{get},\text{set}\}, \partial \mapsto \{\text{get}\}\}}(v)$$

Then supposing $v : \tau$, the cell c has the following type:

$$c \quad : \quad [\text{get} : \text{unit} \rightarrow \tau, \text{set} : \tau \rightarrow \tau]^{\{d:\{\text{get},\text{set}\}, \partial:\{\text{get}\}\}}_{\{\}}$$

Note how the interface is expressed in the type, and how no weakenings show up in the type. However, if we read-weaken c, this information is expressed in the type:

$$\mathbf{weak}_{\{\text{set}\}}(c) \quad : \quad [\text{get} : \text{unit} \rightarrow \tau, \text{set} : \tau \rightarrow \tau]^{\{d:\{\text{get},\text{set}\}, \partial:\{\text{get}\}\}}_{\{\text{set}\}}$$

Given the requirements of the SEND rule, attempting to use the set method of this weakened capability will not be well typed in any context, nor will an attempted set of v returned by reading the weakened capability. This is true even assuming that v is a cell, since weakening information is propagated to τ by the type system, just as weakening is propagated to v by the operational semantics:

$$(\mathbf{weak}_{\{\text{set}\}}(c)).\text{set}(e) \qquad \textit{not well-typed}$$

$$\mathsf{let}\, c' = (\mathbf{weak}_{\{\text{set}\}}(c)).\text{get}()\ \mathsf{in}\ c'.\text{set}(e) \qquad \textit{not well-typed}$$

5.5.2 Ownership types embedding

A language model for alias analysis, together with an *ownership* type analysis, is proposed in [5]. Here we show that their system can be realized in pop (albeit with a use-based security model, rather than the communication-based model of [5]) by choosing an appropriate naming scheme. Assume the following object definition in the language of [5], with the containment relation $p_1 \prec: p_2 \prec: p_3$:

$$[m(x) = e]^{p_2}_{p_1}$$

a similar specification can be defined and statically enforced in pop with the following object definition:

$$[m(x) = e] \cdot p_1 \cdot \{p_1 \mapsto \{m\}, p_2 \mapsto \{m\}, p_3 \mapsto \{m\}\}$$

In general, given any set of contexts $\mathcal{C}$, partial ordering $(\mathcal{C}, \prec:)$ and object o^p_q, we can transform the object into the form $o \cdot q \cdot \varphi$, where $\mathrm{dom}(\varphi) = \{p' \mid p' \prec: p\}$ and for all $p \in \mathrm{dom}(\varphi)$, $\varphi(p)$ is all of o's methods, and carry the transformation recursively through any objects defined in o's methods. Additionally, our type analysis is polymorphic, unlike ownership types, and is thus more flexible.

5.5.3 Classes, `private` and `protected`

By choosing different naming schemes, a variety of security paradigms can be effectively and reliably expressed in pop. One such scheme enforces a strengthened meaning of the `private` and `protected` modifiers in class definitions, a focus of other communication-based capability type analyses [5, 43]. As demonstrated in [43], a `private` field can leak by being returned by reference from a `public` method. Here we show how this problem can be addressed in a use-based model. Assume the following Java-like pseudocode package p, containing class definitions c_1, c_2, and possibly others, where c_2 specifies a method m that leaks a `private` instance variable:

package p begin

class c_1 {
  `public`:
    $f(x) = x$
  `private`:
    $g(x) = x$
  `protected`:
    $h(x) = x$
}

class c_2 {
  `public`:
    $m(x) = b$
    $a =$ new c_1
  `private`:
    $b =$ new c_1
  `protected`:
    $c =$ new c_1
}

$\cdots$

end

We can implement this definition as follows. Interpreting domains as class names in pop, let p denote the set of all class names $c_1, \ldots, c_n$ in package p, and let $p \mapsto \iota$ be syntactic sugar for $c_1 \mapsto \iota_1, \ldots, c_n \mapsto \iota_n$. Then, the appropriate interface for objects in the encoding of class c_1 is as follows:

$$\varphi_1 \quad \triangleq \quad \{p \mapsto \{f, h\}, \partial \mapsto \{f\}\}$$

(Recall that all objects automatically have full access to themselves, so full access for c_1 need not be explicitly stated). The class c_1 can then be encoded as an object *factory*, an object with only one publicly available method that returns new objects in the class, and some arbitrary label d:

$$\begin{aligned} o_1 &\triangleq [f(x) = x,\ g(x) = x,\ h(x) = x] \cdot c_1 \cdot \varphi_1 \\ \mathrm{fctry}_{c_1} &\triangleq [\mathrm{new}(x) = o_1] \cdot d \cdot \{\partial \mapsto \{\mathrm{new}\}\} \end{aligned}$$

To encode c_2, we again begin with the obvious interface definition for objects in the encoding of class c_2:

$$\varphi_2 \quad \triangleq \quad \{p \mapsto \{m, a, c\}\,, \partial \mapsto \{m, a\}\}$$

However, we must now encode *instance variables*, in addition to methods. In general, this is accomplished by encoding instance variables a containing objects as methods $a()$ that return references to objects. Then, any selection of a is encoded as $a().\text{get}()$, and any update with v is encoded $a().\text{set}(v)$. By properly constraining the interfaces on these references, a "Java-level" of modifier enforcement can be achieved; but casting the interfaces of stored objects *extends* the security, by making objects *unusable* outside the intended domain. Let $e \shortmid (\{d_1, \ldots, d_n\}\,, \iota)$ be sugar for $e \shortmid (d_1, \iota) \shortmid \cdots \shortmid (d_n, \iota)$. Using fctry_{c_1}, we may create a `public` version of an object equivalent to o_1, without any additional constraints on its confinement, as follows:

$$o_a \triangleq \text{fctry}_{c_1}.\text{new}()$$

Letting $p' = p - \{c_2\}$, we may create a version of an object equivalent to *o* that is `private` with respect to the encoding of class c_2, using casts as follows:

$$o_b \triangleq (\text{fctry}_{c_1}.\text{new}()) \shortmid (\partial, \varnothing) \shortmid (p', \varnothing)$$

We may create a version of an object equivalent to *o* that is `protected` with respect to the encoding of package p, as follows:

$$o_c \triangleq (\text{fctry}_{c_1}.\text{new}()) \shortmid (\partial, \varnothing)$$

Let o_2 be defined as follows:

$$\begin{aligned} o_2 \quad \triangleq \quad & \mathsf{let}\, r_a = \text{ref}_{\{\partial \mapsto \{\text{set,get}\}\}} o_a \,\mathsf{in} \\ & \mathsf{let}\, r_b = \text{ref}_{\{c_1 \mapsto \{\text{set,get}\}\}} o_b \,\mathsf{in} \\ & \mathsf{let}\, r_c = \text{ref}_{\{c_1 \mapsto \{\text{set,get}\}, p \mapsto \{\text{set,get}\}\}} o_c \,\mathsf{in} \\ & \quad [\, m(x) = \mathbf{s}.b().\text{get}(), \\ & \quad\;\; a(x) = r_a, \\ & \quad\;\; b(x) = r_b, \\ & \quad\;\; c(x) = r_c \,] \cdot c_2 \cdot \varphi_2 \end{aligned}$$

Then fctry_{c_2} is encoded, similarly to fctry_{c_1}, as:

$$\text{fctry}_{c_2} \triangleq [\text{new}(x) = o_2] \cdot d \cdot \{\partial \mapsto \{\text{new}\}\}$$

Given this encoding, if an object stored in b is leaked by a non-local use of m, it is unusable. This is the case because, even though a non-local use of m will return b, in the encoding this return value explicitly states it cannot be used outside the confines of c_2; as a result of the definition of φ_1 and casting, the avatar o_b of b in the encoding has an interface equivalent to:

$$\left\{c_2 \mapsto \{f, h\}, p' \mapsto \varnothing, \partial \mapsto \varnothing\right\}$$

While the communication-based approach accomplishes a similar strengthening of modifier security, the benefits of greater flexibility may be enjoyed via the use-based approach. For example, a `protected` reference can be safely passed outside of a package and then back in, as long as a use of it is not attempted outside the package. Also for example are the fine-grained interface specifications allowed by this approach, enabling greater modifier expressivity— e.g. publicly read-only but privately read/write instance variables.

Conclusion

This thesis has focused on the development of type systems for programming language-based security. We have shown that static type systems are applicable to two distinct security models— the access control model with stack inspection, and the object confinement model. For a consideration of the former, the $\lambda_{\text{sec}}^{\mathsf{S}}$ language was defined, which reflects the low-level behavior of the Java JDK1.2 implementation. $\lambda_{\text{sec}}^{\mathsf{S}}$ uses explicit call-stacks with security annotations, and an explicit stack inspection algorithm for run-time security checks. A monomorphic type system was defined for static enforcement of security in $\lambda_{\text{sec}}^{\mathsf{S}}$, which includes succinct, readable type terms. A type safety theorem implies that run-time stack inspection can be eliminated, but the proof of the theorem was delayed pending development of the λ_{sec} language.

The λ_{sec} language was a re-figuration of $\lambda_{\text{sec}}^{\mathsf{S}}$, with a simpler, more abstract definition of expressions, and a notion of implicit stacks contained in evaluation contexts. This conception of the language is more appealing mathematically, especially for rigorous proof of type safety. By proving that $\lambda_{\text{sec}}^{\mathsf{S}}$ can be simulated in λ_{sec}, confidence was gained in the language's faithfulness to real implementations. A family of polymorphic type systems were developed for λ_{sec}, which was proven safe.

For a consideration of object confinement security, the pop language was defined, an object-based calculus with features for specifying and enforcing object confinement security policies. The pop language is a low-level, flexible system for implementing a variety of higher-level systems. Several examples were discussed, including a class-based languages with strengthened `private` and `protected` modifiers, which prevent leaking of references themselves, rather than merely affecting visibility of instance variable names. A static type discipline for pop was then developed and proven correct; as in the case of λ_{sec}, this type system provides readable declarations of security properties, and type safety implies that run-time security checks can be eliminated.

The type systems for λ_{sec} and pop were both developed using the same methodology; the languages were transformed into the same target language, called pml_B, which is pre-equipped with a sound type system. By proving that these transformations are correct, in that program semantics are preserved, sound indirect type systems were immediately obtained as the composition of the transformations and pml_B type judgements. Direct type systems were also developed, which exploit the transformation and the foundations of the pml_B type system for easy development of soundness proofs, as well as for the design of direct type terms and judgements.

The pml_B language and type system were developed and proven correct by instantiating the HM(X) type and constraint framework with a term language comprising records, sets of atomic elements, and associated operations, and a polymorphic type language comprising row types and conditional constraints. Since the λ_{sec} and pop direct type languages are based on that of pml_B, they reflect the expressivity and notational convenience of row types. Also, since sound implementations of row types and conditional constraints exist, the λ_{sec} and pop type systems benefit from type inference methods. Type safety for pml_B relies on type safety in HM(X); while type safety results do exist for the latter, the first purely syntactic type safety result for HM(X) was provided to ensure a rigorous formal basis for the development of all type systems and associated results, including subject reduction in all cases.

Given these results, the λ_{sec} and pop languages and type systems provide a versatile theoretical foundation for the development of static type disciplines, specifically designed for the specification and enforcement of programming language-based security.

Appendix A

Type System Implementations

```
(**********************************************************)
(*                                                        *)
(* module Seclang: implements \lambda_{sec} language of   *)
(* expressions                                            *)
(*                                                        *)
(**********************************************************)
(* Written by Christian Skalka, Johns Hopkins University 2001 *)
(**********************************************************)

type principal =
    string

type privilege =
    string

type variable =
    string

type expression =
   | Unit
   | Var of variable
   | Fix of variable * variable * expression
   | App of expression * expression
   | Let of variable * expression * expression
   | Enablepriv of privilege * expression
   | Checkpriv of privilege * expression
   | Testpriv of privilege * expression * expression
   | Own of principal * expression
```

```
type phrase =
  | PhraseExpr of expression
  | PhraseLet of variable * expression


(**************************************************************)
(*                                                            *)
(* module localContext: implements a sample local context for *)
(* \lambda_{sec} programs                                     *)
(*                                                            *)
(**************************************************************)
(* Written by Christian Skalka, Johns Hopkins University 2001 *)
(**************************************************************)

(* The initial principal *)
let initp = "_initp"

(* A fixed access credentials mapping. *)
let credentials  = function
  | "root" ->
      [ "disk"; "power"; "memory"; "file"; "thread"; "socket" ]
  | "Joe" ->
      [ "disk" ]
  | "Sue" ->
      [ "power" ]
  | _ ->
      []

(**************************************************************)
(*                                                            *)
(* module Typing: implements \lambda_{sec} S1= type inference *)
(*                                                            *)
(**************************************************************)
(* Written by Christian Skalka, Johns Hopkins University 2001 *)
(**************************************************************)

module System = Herbrand.Make

module type Context = sig

  (* the initial principal in this context *)
  val initp : Seclang.principal

  (* context access control list *)
```

```
  val credentials : Seclang.principal -> Seclang.privilege list

end

module Make (C : Context) = struct

  open System
  open GroundSig
  open Seclang

  type scheme = System.scheme
  type phrase = Seclang.variable * Seclang.expression
  type environment = (Seclang.variable * System.node) list

  (*
     own_rows : privilege list -> node * node
     own_rows [r1,...,rn] returns a pair of rows
     ({r1 : 'a1; ...; rn : 'an; Abs},
      {r1 : 'a1; ...; rn : 'an; rho}
     where 'a1,...,'an and rho are fresh
  *)

  let own_rows rs =
    let rec fr rs =
      match rs with
      | [] ->
        (row_uniform (lo TAbsent), fresh())
      | r::rs' ->
        let (s1, s2) = fr rs' in
        let phi = fresh() in
        (row_component r phi s1, row_component r phi s2)
    in
    let (s1, s2) = fr rs in
    (lo (TSet s1), lo (TSet s2))

  (*
     make_row : privilege -> variable -> variable -> node
     make_row r v1 v2 returns a row {r : v1; v2 }
  *)
  let make_row r v1 v2 = lo (TSet(row_component r v1 v2))

  (* Type inference. *)

  let rec infer p s env = function
```

```
| Unit ->  lo TUnit

| Var x ->

  (* Find the named entry in the current typing
     environment. *)

  let scheme = try
    List.assoc x env
  with Not_found ->
    failwith ("Unbound program variable: " ^ x) in

  (* Instantiate the type scheme. This returns the body of the
     type scheme's instance, and implicitly affects the global
     constraint set. *)

  instantiate scheme

| Fix (z, x, Own (p, e)) ->

  let domain = fresh() in
  let fixt = fresh() in
  let env' = (z, inject fixt) :: (x, inject domain) :: env in
  let (s_abs, s_rho) = own_rows (C.credentials p) in
  let codomain = infer p s_abs env' e in
  unify fixt (lo (arrow domain (lo (arrow s_rho codomain))));
  fixt

| Fix _ ->

  failwith "The body of a function must be signed."

| Own _ ->

  failwith "Signed expressions disallowed in this context"

| App (e1, e2) ->

  let t1 = infer p s env e1 in
  let t2 = infer p s env e2 in
  let alpha = fresh() in
  unify t1 (lo (arrow t2 (lo (arrow s alpha))));
  alpha
```

```
  | Let (x, e1, e2) ->

    (* Infer a type for [e1] and generalize it. Infer a type for
    [e2] within an augmented type environment. *)

    let sigma = infer_and_generalize p s env e1 in
    infer p s ((x, sigma) :: env) e2

  | Testpriv (r, e1, e2) ->

    let rho = fresh() in
    unify (make_row r (fresh()) rho) s;
    let t1 = infer p (make_row r (lo TPresent) rho) env e1 in
    let t2 = infer p (make_row r (lo TAbsent) rho) env e2 in
    unify t1 t2;
    t1

  | Enablepriv (r, e) ->

    if List.mem r (C.credentials p)
    then
        let rho = fresh() in
      unify (make_row r (fresh()) rho) s;
      infer p (make_row r (lo TPresent) rho) env e
    else
      failwith ("User " ^ p ^ " unauthorized for enable " ^ r)

  | Checkpriv (r, e) ->

    try
      unify (make_row r (lo TPresent) (fresh())) s;
      infer p s env e
    with Inconsistency ->
      failwith ("Resource [" ^ r ^ "] is unauthorized")

and infer_and_generalize p s env e =

  (* Infer a type for [e], making sure that all variables
     freshly created in this sub-derivation are marked as
     such, i.e. would be quantified by a $(\exists Intro)$
     rule. Generalize the type thus obtained. *)

  scope (fun () ->
```

```
      generalize (infer p s env e)
    )

  (* External interface. *)

  let run env (x, e) =
    let rec initrow rs =
      match rs with
      | [] ->
        row_uniform (lo TAbsent)
      |      r::rs' ->
        row_component r (lo TPresent) (initrow rs')
    in
    let s = (lo (TSet(initrow (C.credentials C.initp)))) in
    x, infer_and_generalize C.initp s env e

end

(***********************************************************)
(*                                                         *)
(* module Hmx: implements HM(X) type inference             *)
(*                                                         *)
(***********************************************************)
(*              Written by Francois Pottier                *)
(*         projet Cristal, INRIA Rocquencourt              *)
(*                                                         *)
(*    Copyright 2002 Institut National de Recherche en     *)
(*              Informatique et Automatique.               *)
(***********************************************************)

module type S = sig

  (* Names of primitive operations. *)

  type primitive

  (* Type variables. *)

  type variable

  (* Type schemes. *)

  type scheme
```

```
  (* Type environments. *)

  type environment =
      (string * scheme) list

  (* Program terms. A phrase is a single
     toplevel \texttt{let} definition. *)

  type expression =
    | Prim of primitive
    | Var of string
    | Fix of string * string * expression
    | App of expression * expression
    | Let of string * expression * expression

  type phrase =
      string * expression

  (* Type inference. This function accepts an environment and
     a phrase. It infers a type, generalizes it, and returns
     it, together with the name of the variable being defined. *)

  val run: environment -> phrase -> string * scheme

end

module Make
    (G : Ground.Signature)
    (X : ConstraintSystem.S with type 'a preterm = 'a G.term)
    (P : Primitives.S with type scheme = X.scheme)
    : S with type primitive = P.name
         and type variable = X.variable
         and type scheme = X.scheme

module Make
    (G : Ground.Signature)
    (X : ConstraintSystem.S with type 'a preterm = 'a G.term)
    (P : Primitives.S with type scheme = X.scheme)
= struct

  (* Names of primitive operations. *)

  type primitive = P.name
```

```
(* Type variables. *)

type variable = X.variable

(* Type schemes. *)

type scheme = X.scheme

(* Type environments. *)

type environment =
    (string * scheme) list

(* Program terms. A phrase is a single toplevel
   let definition. *)

type expression =
  | Prim of primitive
  | Var of string
  | Fix of string * string * expression
  | App of expression * expression
  | Let of string * expression * expression

type phrase =
    string * expression

(* Type inference. *)

let rec infer env = function
  | Prim name ->

    (* Look up the named primitive, and instantiate its type
         scheme. *)

    X.instantiate (P.map name)

  | Var x ->

    (* Find the named entry in the current typing
       environment. *)

    let scheme = try
      List.assoc x env
    with Not_found ->
```

```
        failwith ("Unbound program variable: " ^ x) in

      (* Instantiate the type scheme. This returns the body of the
         type scheme's instance, and implicitly affects the global
         constraint set. *)

      X.instantiate scheme

    | Fix (z, x, e) ->

      let domain = X.fresh() in
      let fixt = X.fresh()
      let codomain =
            infer ((z, X.inject fixt) ::
                   (x, X.inject domain) :: env) e
        in
      X.constrain fixt (G.arrow domain codomain);
      lo (G.arrow domain codomain);

    | App (e1, e2) ->

      let alpha = X.fresh() in
      X.constrain (infer env e1) (G.arrow (infer env e2) alpha);
      alpha

    | Let (x, e1, e2) ->

      (* Infer a type for [e1] and generalize it. Infer a type
           for [e2] within an augmented type environment. *)

      infer ((x, infer_and_generalize env e1) :: env) e2

and infer_and_generalize env e =

    (* Infer a type for [e], making sure that all variables
       freshly created in this sub-derivation are marked as
       such, i.e. would be quantified by a ($\exists$ Intro)
       rule. Generalize the type thus obtained. *)

    X.scope (fun () ->
      X.generalize (infer env e)
    )

(* External interface. *)
```

```
  let run env (x, e) =
    x, infer_and_generalize env e
end

(**********************************************************)
(*                                                        *)
(* signature ConstraintSystem: describes the expected     *)
(* form of an instance of HM(X)                           *)
(*                                                        *)
(**********************************************************)
(*              Written by Francois Pottier               *)
(*         projet Cristal, INRIA Rocquencourt             *)
(*                                                        *)
(*    Copyright 2002 Institut National de Recherche en    *)
(*              Informatique et Automatique.              *)
(**********************************************************)

module type S = sig

  type variable

  type 'a preterm

  type term =
      variable preterm

  type scheme

  (* [fresh()] returns a fresh variable. *)

  val fresh: unit -> variable

  (* [lo term] returns a fresh variable. It implicitly
     makes [term] its lower bound in the global constraint
     set.  [hi term] returns a fresh variable. It implicitly
     makes [term] its upper bound in the global constraint set. *)

  val lo: term -> variable
  val hi: term -> variable

  (* [row_component l v1 v2] returns a fresh variable,
     implicitly equated with the row (l: v_1; v_2).
     [row_uniform v] returns a fresh variable, implicitly
```

```
   equated with the row \delta(v). *)

val row_component: string -> variable -> variable -> variable
val row_uniform: variable -> variable

(* [constrain v term] adds a subtyping constraint
   between [v] and [term] to the global constraint set.
   The exception [Inconsistency] is raised if the constraint
   set becomes inconsistent as a result of this addition. *)

exception Inconsistency

val constrain: variable -> term -> unit

(* [scope action] executes the specified [action], with the
   side effect that all variables freshly created during its
   scope are marked as such. *)

val scope: (unit -> 'a) -> 'a

(* [generalize v] creates a type scheme out of the constraints
   created during the current invocation of [exists], whose
   entry point is assumed to be [v]. *)

val generalize: variable -> scheme

(* [instantiate scheme] creates a fresh instance of the type
   scheme [scheme]. It returns its entry point, and implicitly
   affects the global constraint set. *)

val instantiate: scheme -> variable

(* [inject v] turns a type variable into a (trivial) type
   scheme. *)

val inject: variable -> scheme

(* Printing terms. *)

module Print : sig

  (* [reset] resets the mechanism which assigns new names to
     type variables. *)
```

```
    val reset: unit -> unit

    (* [variable v] prints a type variable, together with
       the constraints bearing on it. [scheme] prints a type
       scheme. *)

    val variable: variable -> string
    val scheme: scheme -> string

  end
end

(**********************************************************)
(*                                                        *)
(* signature Primitives: describes the expected form of   *)
(* type scheme and primitive constant binding             *)
(* implementations for instances of HM(X)                 *)
(*                                                        *)
(**********************************************************)
(*              Written by Francois Pottier               *)
(*         projet Cristal, INRIA Rocquencourt             *)
(*                                                        *)
(*    Copyright 2002 Institut National de Recherche en    *)
(*              Informatique et Automatique.              *)
(**********************************************************)

module type S = sig

  (* Names of primitive operations. *)

  type name

  (* Type schemes. *)

  type scheme

  (* A mapping between the two. *)

  val map: name -> scheme

end
```

```
(**********************************************************)
(*                                                        *)
(* signature Ground: describes the operations which       *)
(* abstractly represent a free term algebra               *)
(*                                                        *)
(**********************************************************)
(*              Written by Francois Pottier               *)
(*         projet Cristal, INRIA Rocquencourt             *)
(*                                                        *)
(*    Copyright 2002 Institut National de Recherche en    *)
(*              Informatique et Automatique.              *)
(**********************************************************)

module type Signature = sig

  (* The type of terms. *)

  type 'a term

  (* Abstract operations on terms. *)

  exception Iter2

  val arity: 'a term -> int
  val map: ('a -> 'b) -> 'a term -> 'b term
  val fork: ('a -> 'b * 'c) -> 'a term -> 'b term * 'c term
  val iter: ('a -> unit) -> 'a term -> unit
  val fold: ('a -> 'b -> 'b) -> 'a term -> 'b -> 'b
  val iter2: ('a -> 'b -> unit) -> 'a term -> 'b term -> unit

  (* The type of symbols, i.e. head constructors of terms. *)

  type symbol

  val matches: symbol -> 'a term -> bool
  val sprint: symbol -> string

  (* The type of labels, used to name every argument of every
     type constructor. *)

  type label

  (* [print term] returns a list of labeled sub-terms and
```

```
     tokens.  [parenthesize label subterm] tells whether the
     given [subterm] must be parenthesized, if found at the
     given [label] within a larger term. [safe label] tells
     whether a label occurs between two tokens, i.e. subterms
     at this label \emph{never} need to be parenthesized. *)

  val print: 'a term -> (label * 'a) Tree.element list
  val parenthesize: label -> 'a term -> bool
  val safe: label -> bool

  (* Injections into terms. These are provided for use by the
     typechecker -- constraint generation would be impossible
     if terms were entirely abstract! *)

  val arrow: 'a -> 'a -> 'a term

end
```

Bibliography

[1] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In G. Tsudik, editor, *Proceedings of the 6th Conference on Computer and Communications Security*, Singapore, November 1999. ACM Press. To appear.

[2] Borris Bokowski and Jan Vitek. Confined types. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on ObjectOriented Programming Systems, Languages, and Applications (OOPSLA)*, November 1999.

[3] John Boyland, James Noble, and William Retert. Capabilities for aliasing: A generalisation of uniqueness and read-only. In *ECOOP'01 — Object-Oriented Programming, 15th European Conference*, Lecture Notes in Computer Science, Berlin, Heidelberg, New York, 2001. Springer.

[4] Ciaran Bryce and Jan Vitek. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[5] David Clarke, James Noble, and John Potter. Simple ownership types for object containment. In *ECOOP'01 — Object-Oriented Programming, 15th European Conference*, Lecture Notes in Computer Science, Berlin, Heidelberg, New York, 2001. Springer.

[6] Karl Crary and Stephanie Weirich. Resource bound certification. In *Symposium on Principles of Programming Languages*, 2000.

[7] D. Denning. A lattice model of secure information flow. In *Communications of the ACM*, pages 236–243. ACM, May 1976.

[8] Mark Miller *et. al.* The E programming language. `http://www.erights.org`.

[9] Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. In *Proceedings of the 29th Symposium on Principles of Programming Languages (POPL'02)*, January 2002.

[10] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, December 1997.

[11] Li Gong. Java security architecture (JDK1.2). `http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html`, October 1998.

[12] Norman Hardy. The KeyKOS architecture. *Operating Systems Review*, pages 8–25, October 1985.

[13] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *1998 USENIX Annual Technical Conference*, pages 259–270, New Orleans, LA, 1998.

[14] C. Hawblitzel and T. von Eicken. A case for language-based protection. Technical Report TR98-1670, Cornell University, 1998.

[15] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, San Diego, California, January 1998.

[16] Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *Proc. 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 81–92, Portland, OR, 2002.

[17] Richard Y. Kain and Carl E. Landwehr. On access checking in capability-based systems. *IEEE Transactions on Software Engineering*, 13(2):202–207, February 1987.

[18] J. Lassez, M. Maher, and K. Marriott. *Unification Revisited.* in J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kauffman, 1987.

[19] SUN Microsystems. Java 2 SDK, standard edition documentation. New API for privileged blocks, `http://java.sun.com/products/jdk/1.2/docs/guide/security/doprivileged.html`.

[20] D. Naumann and A. Banerjee. A static analysis for instance-based confinement in java. Submitted Nov. 2001.

[21] D. Naumann and A. Banerjee. Representation independence, confinement, and access control. In *Proceedings of the 29th Symposium on Principles of Programming Languages (POPL'02)*, 2002.

[22] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, October 1996. USENIX.

[23] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[24] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.

[25] François Pottier. A semi-syntactic soundness proof for HM(X). Research Report 4150, INRIA, March 2001.

[26] François Pottier. A simple view of type-secure information flow in the π-calculus. In *Proceedings the 15th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, June 2002.

[27] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, September 2000.

[28] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer Verlag, April 2001.

[29] Didier Rémy. Extending ML type system with a sorted equational theory. Technical Report 1766, INRIA, Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France, 1992.

[30] Didier Rémy. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, New-York, 1992. ACM Press.

[31] Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, INRIA, 1993.

[32] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[33] Didier Rémy. Typing record concatenation for free. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[34] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. In *Proc. 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 217–232, Portland, OR, 2002.

[35] Jonathan Shapiro, Jonathan Smith, and David Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.

[36] Jonathan Shapiro and Samuel Weber. Verifying the EROS confinement mechanism. In *21st IEEE Computer Society Symposium on Research in Security and Privacy*, 2000.

[37] Christian Skalka and Scott Smith. Static enforcement of security with types. In *Proceedings of the the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 34–45, Montréal, Canada, September 2000.

[38] Christian Skalka and Scott Smith. Set types and applications. In *Workshop on Types in Programming (TIP02)*, 2002.

[39] Martin Sulzmann. Proofs of Soundness and Completeness of Type Inference for HM(X). Research Report YALEU/DCS/RR-1102, Yale University, Department of Computer Science, February 1997.

[40] Martin Sulzmann. *A general framework for Hindley/Milner type systems with constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.

[41] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science, July 1999.

[42] L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure network objects. In *IEEE Symposium on Security and Privacy*, May 1996.

[43] Jan Vitek and Boris Bokowski. Confined types in java. *Software—Practice and Experience*, 31(6):507–532, May 2001.

[44] David Walker. A type system for expressive security policies. In *Twenty-seventh Symposium on Principles of Programming Languages*, pages 254–267, Boston, MA, January 2000. ACM SIGPLAN.

[45] Dan S. Wallach. *A New Approach to Mobile Code Security*. PhD thesis, Princeton University, January 1999.

[46] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.

[47] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

[48] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor. *Communications of the ACM*, 17(6):337–345, 1974.

Vita

Christian Skalka was born in New York City on February 17, 1969. He spent most of his youth in the NYC metro area and Vermont, where he ski raced and attended high school at the Green Mountain Valley School. Christian received a BA from St. John's College in May 1991, where he studied classical philosophy and mathematics. He worked on the GenBank genetics database at Los Alamos National Laboratories and the National Institutes of Health from 1992 until 1995, and earned an MS in Logic, Computation and Methodology from Carnegie Mellon University, Department of Philosophy, in May 1997. Christian's MS thesis, *Some Decision Problems for ML Refinement Types*, was written under the guidance of Frank Pfenning. His doctoral research with Scott Smith at Johns Hopkins began immediately thereafter. Currently, he is an Assistant Professor of Computer Science at the University of Vermont.