Safety Guarantees from Explicit Resource Management

David Aspinall, Patrick Maier, and Ian Stark

Laboratory for Foundations of Computer Science
School of Informatics, The University of Edinburgh, Scotland
{David.Aspinall,Patrick.Maier,Ian.Stark}@ed.ac.uk

Abstract. We present a language and a program analysis that certifies the safe use of flexible resource management idioms, in particular advance reservation or "block booking" of costly resources. This builds on previous work with *resource managers* that carry out runtime safety checks, by showing how to assist these with compile-time checks. We give a small ANF-style language with explicit resource managers, and introduce a type and effect system that captures their runtime behaviour. In this setting, we identify a notion of *dynamic safety* for running code, and show that dynamically safe code may be executed without runtime checks. We show a similar *static safety* property for type-safe code, and prove that static safety implies dynamic safety. The consequence is that typechecked code can be executed without runtime instrumentation, and is guaranteed to make only appropriate use of resources.

1 Introduction

Safe management of resources is a crucial aspect of software correctness. Bad resource management impacts reliability and security. The more expensive a resource or the more complex its usage pattern, the more important is good management. For example, a media player could crash badly, leaving the hardware in a messy state, if its memory management was governed by the overly optimistic assumption that every request for memory will succeed. Malware on a mobile phone can defraud an unaware user by maliciously sending text messages to premium rate numbers, if there is no effective management of network access [12]. On current mobile platforms such as Java MIDP 2.0, management of network access is commonly left to the user, but users can easily be deceived by social engineering attacks.

Unfortunately, current programming languages do not provide special mechanisms for resource management. Therefore, programmers can only hope that their applications are resource safe, or use necessarily imprecise analyses to try to show this. For example, there are type systems that over-approximate (hopefully tightly) the memory requirements of an application [6], and static analyses that over-approximate the number of text messages being sent by an application [7].

These approaches may fail if a dynamic set of resources must be managed, as with *bulk messaging* where the user wants to send a text message to a number of recipients selected from an address book. Because of the cost of sending text messages, the user must authorise each recipient (i. e., their phone number) explicitly. This could happen individually, just before each message is being sent, or collectively, before sending the first message. Collective authorisation, or *block booking* of resources, is preferable but requires detailed resource management, keeping track of the (multi-)set of authorised resources – in this case the permitted phone numbers.

In this paper, we present a language-based mechanism that provides programmers with a safe way to control complex resource usage patterns using a notion of *resource manager*. Figure 1 shows the code of a bulk messaging application using resource managers in our intermediate-level functional programming language. The language and functions used will be explained in full detail in Section 2; for now, we just give an outline of operation. The function send_bulk calls send_msgs to send the message msg to the phone numbers stored in the array nums. Along with these two arguments send_msgs takes a resource manager m' which encapsulates the resources that have been authorised (during the call to enable) to send the messages. For each phone number in nums, send_msgs calls the wrapper function send_msg, passing along a resource manager. Prior to calling the primitive send function prim_send_msg, the wrapper checks (using assertAtLeast) whether its input manager m contains the resource required to send a message to num; if the resource is not present, the program will abort with a runtime error, otherwise send_msg removes the resource from the manager (using split), and returns the modified manager as m'.

The bulk messaging application is (dynamically) resource safe by construction, as the resource managers will trap attempts to abuse resources. The resource manager abstraction works in tandem with a static analysis, so that programs which can be proved resource safe statically can be treated more efficiently at runtime by removing the dynamic accounting code. In Section 3.2, we prove resource safety statically for the bulk messaging application.

Our contribution is two-fold. In Section 2, we develop a functional programming language for coding complex resource idioms, such as block booking resources in the bulk messaging application. The language is essentially a first-order functional language in administrative normal form (ANF) [10] with a novel type system serving two purposes. First, the type system names input and output parameters of functions and avoids shadowing of previously bound names, thus admitting to view functions as relations (expressed by logical formulae) between their input and output parameters. Second, the language includes a special, linear type for resource managers, where linearity serves as a means of introducing stateful objects into an otherwise pure functional language. Resource managers track what resources a program is allowed to use, and the operational semantics causes the program to go wrong (i. e., abort with a runtime error) as soon as it attempts to abuse resources. This induces a notion of *dynamic resource safety*, which holds if a program never attempts to abuse resources. In this case, accounting is not necessary. As our first result, we show that erasing resource managers does not alter the semantics of dynamically resource safe programs.

Decisions about which resources programs may use are typically guided by *resource policies*. From the point of view of a program, a policy is simply an oracle determining what resources to grant; and we abstract this as a non-deterministic operation on resource managers. This covers many concrete policy mechanisms, both static (e. g., Java-style policy files) or dynamic (e. g., user interaction); see [3] for more on the interaction of resource managers and policies.

```
send_bulk ::
λ let (r) = res_from_nums (nums) in
   let (m) = init () in
   let (m',r') = enable (m,r) in
   let (n) = size (r') in
   if n then let () = consume (m') in
             ret ()
        else  let (m'') = send_msgs (msg,nums,m') in
              let (m''') = assertEmpty (m'') in
              let () = consume (m''') in
              ret () :
(msg:str, nums:str[]) → ()

res_from_nums ::
λ let (i) = length (nums) in
   let (r) = empty () in
   let (r') = res_from_nums' (nums,r,i) in
   ret (r') :
(nums:str[]) → (r':res{})

res_from_nums' ::
λ if i then let (i') = sub (i,1) in
            let (num) = read (nums,i') in
            let (c) = fromstr (num) in
            let (r_c) = single (c,1) in
            let (r'') = sum (r, r_c) in
            let (r') = res_from_nums' (nums,r'',i') in
            ret (r')
       else let (r') = id (r) in
            ret (r') :
(nums:str[], r:res{}, i:int) → (r':res{})
```

```
send_msgs ::
λ let (i) = length (nums) in
   let (m') = send_msgs' (msg,nums,m,i) in
   ret (m') :
(msg:str, nums:str[], m:mgr) → (m':mgr)

send_msgs' ::
λ if i then let (i') = sub (i,1) in
            let (num) = read (nums,i') in
            let (m'') = send_msg (msg,num,m) in
            let (m') = send_msgs' (msg,nums,m'',i') in
            ret (m')
       else let (m') = id (m) in
            ret (m') :
(msg:str, nums:str[], m:mgr, i:int) → (m':mgr)

send_msg ::
λ let (c) = fromstr (num) in
   let (r) = single (c,1) in
   let (m',m_r) = split (m,r) in
   let (m_r') = assertAtLeast (m_r,r) in
   let () = prim_send_msg (msg,num) in
   let () = consume (m_r') in
   ret (m') :
(msg:str, num:str, m:mgr) → (m':mgr)

prim_send_msg ::
λ . . . :
(msg:str, num:str) → ()
```

Fig. 1. Bulk messaging application.

In Section 3 we present our second contribution, an effect type system for deriving relational approximations of functions. These approximations are expressed as pairs of constraints in a first-order logic, specifying a pre- and postcondition (or rather, state transforming action) of a given function, similar to Hoare type theory [11]; note that the use of logical formulae as effects is the rationale behind choosing a programming language where functions have named input and output parameters. Typability of functions in the effect type system induces a notion of *static resource safety*. As our second result, we prove a soundness theorem stating that static implies dynamic resource safety. As a corollary, we show that resource managers can always be erased from statically resource safe programs. Proofs have been omitted due to lack of space.

2 A Programming Language for Resource Management

We introduce a simple programming language with built-in constructs for handling resource managers. The language is essentially a simply-typed first-order functional language in ANF [10], with the additional features that functions take and return tuples of values, function types name input and output arguments, scoping avoids shadowing, and the type of resource managers enforces a linearity restriction on its values. The first three of these features are related to giving the language a relational appeal: for the purpose of specifying and reasoning logically, functions ought to be viewed as relations

$$
\begin{array}{rcll}
\langle\text{fundecl}\rangle & ::= & \langle\text{prodtype}\rangle \longrightarrow \langle\text{prodtype}\rangle & \textit{(built-in function)}\\
 & | & \boldsymbol{\lambda}\langle\text{exp}\rangle\ \mathbf{:}\ \langle\text{prodtype}\rangle \longrightarrow \langle\text{prodtype}\rangle & \textit{(}\lambda\textit{-abstraction)}\\
\langle\text{exp}\rangle & ::= & \mathbf{if}\ \langle\text{val}\rangle\ \mathbf{then}\ \langle\text{exp}\rangle\ \mathbf{else}\ \langle\text{exp}\rangle & \textit{(conditional)}\\
 & | & \mathbf{let}\ (\langle\text{var}\rangle,\ldots,\langle\text{var}\rangle) = \langle\text{fun}\rangle\ (\langle\text{val}\rangle,\ldots,\langle\text{val}\rangle)\ \mathbf{in}\ \langle\text{exp}\rangle & \textit{(function call)}\\
 & | & \mathbf{ret}\ (\langle\text{var}\rangle,\ldots,\langle\text{var}\rangle) & \textit{(return)}\\
\langle\text{val}\rangle & ::= & \langle\text{const}\rangle \mid \langle\text{var}\rangle & \\
\langle\text{prodtype}\rangle & ::= & (\langle\text{var}\rangle\mathbf{:}\langle\text{type}\rangle,\ldots,\langle\text{var}\rangle\mathbf{:}\langle\text{type}\rangle) & \\
\langle\text{type}\rangle & ::= & \langle\text{datatype}\rangle \mid \mathbf{mgr} & \\
\langle\text{datatype}\rangle & ::= & \mathbf{unit} \mid \mathbf{int} \mid \mathbf{str} \mid \mathbf{res} \mid \mathbf{res\{\}} \mid \langle\text{datatype}\rangle\,[\,] &
\end{array}
$$

Fig. 2. BNF grammar.

between input and output parameters. The fourth feature is a means of introducing state into a functional language.

The choice for such a language has been inspired by Grail [2], another first-order functional language in ANF. Moreover, Appel [1] argues that ANF, the intermediate language used by many compilers for functional languages, and SSA, the intermediate representation used by most compilers for imperative languages, are essentially the same thing. Therefore, our language should capture the essence of first-order programming languages, whether functional or imperative.

2.1 Syntax and Static Semantics

Grammar. Figure 2 shows the grammar of the programming language. The nonterminals ⟨fun⟩, ⟨var⟩ and ⟨const⟩ represent *functions*, *variables* and *constants*, respectively. A *program* $\mathit{\Pi}$ is a partial function from ⟨fun⟩ to ⟨fundecl⟩, i. e., $\mathit{\Pi}$ maps functions to function declarations, which are either type declarations for built-in functions or λ-abstractions (with type annotations serving as variable binders). We use the notation $\mathit{\Pi}(f) = [\boldsymbol{\lambda}\ldots]\sigma \longrightarrow \sigma'$ if we are only interested in the type of f, regardless whether f is built-in or a λ-abstraction. By $dom(\mathit{\Pi})$, we denote the domain of $\mathit{\Pi}$. We denote the restriction of $\mathit{\Pi}$ to the built-in functions by $\mathit{\Pi}_0$, i. e., $\mathit{\Pi}(f)$ is a λ-abstraction if and only if $f \in dom(\mathit{\Pi}) \setminus dom(\mathit{\Pi}_0)$. We assume that $\mathit{\Pi}_0$ declares exactly the functions that are shown in Figure 4.

The grammar of *expressions* $e \in$ ⟨exp⟩ and *values* $v \in$ ⟨val⟩ is quite standard for a first-order functional language in ANF. Throughout, functions operate on tuples of values, which is reflected by the syntax for function call and return. The sets of free and bound (by the let-construct) variables of an expression e, denoted by $\mathit{free}(e)$ and $\mathit{bound}(e)$ respectively, are defined in the usual way.

Datatypes $\tau \in$ ⟨datatype⟩ comprise the unit type, integers, strings, resources, multisets of resources, and arrays. A *type* $\tau \in$ ⟨type⟩ is either a datatype or the special type of resource managers, denoted $\mathbf{mgr}$. See Section 2.2 for the interpretations of types. A tuple $(x_1\mathbf{:}\tau_1,\ldots,x_n\mathbf{:}\tau_n) \in$ ⟨prodtype⟩ is a *product type* if the variables $x_1, \ldots, x_n$ are pairwise distinct. Product types appear to associate types to variables, but they really associate variables *and* types to positions in tuples. A pair of product types of the form $(x_1{:}\tau_1,\ldots,x_m{:}\tau_m) \longrightarrow (x'_1{:}\tau'_1,\ldots,x'_n{:}\tau'_n)$ forms a *function type* if the variable sets $\{x_1,\ldots,x_m\}$ and $\{x'_1,\ldots,x'_n\}$ are disjoint. We call the product types to the left and right of the arrow *argument type* and *return type*, respectively. As an example consider the type of the function send_msg from Figure 1. It states that send_msg takes two strings and a resource manager and returns a resource manager, while at the same time binding the names of the formal input parameters msg, num and m and announcing that the formal output parameter will be m'.

Static typing. A *type environment* Γ is a functional association list of type declarations of the form $x{:}\tau$, where x is a variable and τ a type. Being functional implies that whenever Γ contains two type declarations $x{:}\tau$ and $x{:}\tau'$ we must have $\tau = \tau'$. Therefore, Γ can be seen as a partial function mapping variables to types. By $dom(\Gamma)$, we denote the domain of this partial function, and for $x \in dom(\Gamma)$, we may write $\Gamma(x)$ for the unique type which Γ associates to x. We write type environments as comma-separated lists, the empty list being denoted by $\emptyset$. The restriction $\Gamma|_X$ of Γ to a set of variables X, is defined in the usual way and induces a partial order $\succeq$ type environments, where $\Gamma' \succeq \Gamma$ iff $\Gamma'|_{dom(\Gamma)} = \Gamma$.

We call a type environment $\Gamma = x_1{:}\tau_1,\ldots,x_n{:}\tau_n$ *linear* if the variables $x_1,\ldots,x_n$ are pairwise distinct. Note that such a linear type environment Γ may be viewed as a product type $\sigma = (x_1{:}\tau_1,\ldots,x_n{:}\tau_n)$, and vice versa. Occasionally, we will write $\Pi(f) = [\boldsymbol{\lambda}\ldots]\Gamma \longrightarrow \Delta$ to emphasise that argument and return types of the function f are to be viewed as linear type environments.

Figure 3 shows the typing rules for the programming language. The judgement $C;\Gamma \vdash v : \tau$ expresses that the value v has type τ in type environment Γ and context C, where a *context* is a set of variables (generally the set of variables occurring in some super-expression of v). Note that (T-const) restricts program constants to the unit value, integers and strings, which are the interpretations of the types $\mathbf{unit}$, $\mathbf{int}$ and $\mathbf{str}$, respectively (see Section 2.2). All other types are abstract in the sense that their values can only be accessed through built-in functions.

The judgement $C;\Gamma \vdash_\Pi e : \sigma$ means that the expression e has product type σ in type environment Γ, context C and program Π. If the program is understood we may write $C;\Gamma \vdash e : \sigma$. There are three things worth noting about expression typing. First, although the type system is linear, weakening and contraction are available to all types but $\mathbf{mgr}$, rendering $\mathbf{mgr}$ the sole linear type of the language. Second, the side condition of (T-let) ensures that let-bound variables do not shadow any variables in the context (which is generally a superset of the set of variables occurring in the let-expression). Third, the rule (T-ret) matches the variables in the return expression to the variables in the product type, thus enforcing that an expression uniformly uses the same variables to return its results (even though these return variables may be let-bound in different branches of the expression). Note that (T-ret) is the only rule to exploit type information about variables. Finally, the judgement $\Gamma \vdash e : \sigma$ (or $\Gamma \vdash_\Pi e : \sigma$ if we want to stress the program Π) means that e has product type σ in a linear type environment Γ.

The judgement $\Pi \vdash f$ states that f is a well-typed λ-abstraction in program Π. Note that the syntax of λ-abstractions does not appear to bind variables, yet it does bind the variables hidden in the argument type. Note also that the restriction on function

Typing of values $C; \mathit{\Gamma} \vdash v : \tau$

$$\text{(T-var)}\ \frac{}{C; x{:}\tau \vdash x : \tau}\ \text{if } x \in C \qquad \text{(T-const)}\ \frac{}{C; \emptyset \vdash d : \tau}\ \text{if } \begin{cases} d \in \tau \wedge \\ \tau \in \{\mathbf{unit}, \mathbf{int}, \mathbf{str}\} \end{cases}$$

Typing of expressions $C; \mathit{\Gamma} \vdash e : \sigma$

$$\text{(T-weak)}\ \frac{C; \mathit{\Gamma} \vdash e : \sigma}{C; \mathit{\Gamma}, x{:}\tau \vdash e : \sigma}\ \text{if } \begin{cases} x \in C \wedge \\ \tau \neq \mathbf{mgr} \end{cases} \qquad \text{(T-contr)}\ \frac{C; \mathit{\Gamma}, x{:}\tau, x{:}\tau \vdash e : \sigma}{C; \mathit{\Gamma}, x{:}\tau \vdash e : \sigma}\ \text{if } \tau \neq \mathbf{mgr}$$

$$\text{(T-if)}\ \frac{C; \mathit{\Gamma} \vdash v : \mathbf{int} \qquad C; \mathit{\Gamma}' \vdash e_1 : \sigma \qquad C; \mathit{\Gamma}' \vdash e_2 : \sigma}{C; \mathit{\Gamma}, \mathit{\Gamma}' \vdash \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \sigma} \qquad \text{(T-xch)}\ \frac{C; \mathit{\Gamma}, \mathit{\Gamma}' \vdash e : \sigma}{C; \mathit{\Gamma}', \mathit{\Gamma} \vdash e : \sigma}$$

$$\text{(T-ret)}\ \frac{C; \mathit{\Gamma}_1 \vdash x_1 : \tau_1 \quad \ldots \quad C; \mathit{\Gamma}_n \vdash x_n : \tau_n}{C; \mathit{\Gamma}_1, \ldots, \mathit{\Gamma}_n \vdash \mathbf{ret}\ (x_1, \ldots, x_n) : (x_1{:}\tau_1, \ldots, x_n{:}\tau_n)}$$

$$\text{(T-let)}\ \frac{\begin{array}{c} \mathit{\Pi}(f) = [\boldsymbol{\lambda} \ldots](z_1{:}\tau_1, \ldots, z_m{:}\tau_m) \longrightarrow (z_1'{:}\tau_1', \ldots, z_n'{:}\tau_n') \\ C; \mathit{\Gamma}_1 \vdash v_1 : \tau_1 \quad \ldots \quad C; \mathit{\Gamma}_n \vdash v_m : \tau_m \\ C \cup \{x_1', \ldots, x_n'\}; \mathit{\Gamma}', x_1'{:}\tau_1', \ldots, x_n'{:}\tau_n' \vdash e' : \sigma'' \end{array}}{C; \mathit{\Gamma}_1, \ldots, \mathit{\Gamma}_m, \mathit{\Gamma}' \vdash \mathbf{let}\ (x_1', \ldots, x_n') = f\ (v_1, \ldots, v_m)\ \mathbf{in}\ e' : \sigma''}\ \text{if } (*)$$

$$\text{where } (*) \begin{cases} x_1', \ldots, x_n' \text{ pairwise distinct} \wedge \\ x_1', \ldots, x_n' \notin C \cup \mathit{dom}(\mathit{\Gamma}') \end{cases}$$

Typing of expressions $\mathit{\Gamma} \vdash e : \sigma$

$$\text{(T-lin)}\ \frac{\mathit{dom}(\mathit{\Gamma}); \mathit{\Gamma} \vdash e : \sigma}{\mathit{\Gamma} \vdash e : \sigma}\ \text{if } \mathit{\Gamma} \text{ linear}$$

Well-typedness of λ-abstractions $\mathit{\Pi} \vdash f$

$$\text{(T-lam)}\ \frac{\begin{array}{c} \mathit{\Pi}(f) = \boldsymbol{\lambda} e : (x_1{:}\tau_1, \ldots, x_m{:}\tau_m) \longrightarrow \sigma' \\ x_1{:}\tau_1, \ldots, x_m{:}\tau_m \vdash e : \sigma' \end{array}}{\mathit{\Pi} \vdash f}$$

Fig. 3. Typing rules (for a fixed program $\mathit{\Pi}$).

types means that the return variables of the body of a λ-abstraction must be disjoint from its argument variables. Finally, we call a program $\mathit{\Pi}$ *well-typed* if $\mathit{\Pi} \vdash f$ for all $f \in \mathit{dom}(\mathit{\Pi}) \setminus \mathit{dom}(\mathit{\Pi}_0)$.

Lemma 1. *Let e be an expression (referring to an implicit program $\mathit{\Pi}$), $\mathit{\Gamma}$ a type environment and σ a product type.*

1. *If $\mathit{\Gamma} \vdash e : \sigma$ then $\mathit{free}(e) \subseteq \mathit{dom}(\mathit{\Gamma})$ and $\mathit{bound}(e) \cap \mathit{dom}(\mathit{\Gamma}) = \emptyset$.*
2. *If $\mathit{\Gamma} \vdash e : \sigma$ and $X \supseteq \mathit{free}(e)$ then $\mathit{\Gamma}|_X \vdash e : \sigma$.*

2.2 Interpretation of Types and Effects of Built-in Functions

Constraints. To provide a formal semantics for the built-in functions, we introduce a many-sorted first-order language $\mathcal{L}$ with equality. Sorts of $\mathcal{L}$ are the datatypes of the programming language (note that this excludes the type $\mathbf{mgr}$). Formulae of $\mathcal{L}$ are formed from atomic formulae using the usual Boolean connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$ (in decreasing order of precedence), and the quantifiers $\forall x{:}\tau$ and $\exists x{:}\tau$, where $x \in \langle \text{var} \rangle$

is a variable and $\tau \in \langle\text{datatype}\rangle$ a sort. Atomic formulae are the Boolean constants $\top$ and $\bot$, or are constructed from terms using the binary equality predicate $\approx$ (which is available for all sorts), the binary inequality predicate $\leq$ on sort $\mathbf{int}$ or the binary inclusion predicate $\subseteq$ on sort $\mathbf{res\{\}}$. Terms are constructed from variables in $\langle\text{var}\rangle$ and the term constructors, which are introduced below, alongside associating the sorts to specific interpretations.

Sort $\mathbf{unit}$ is interpreted by the one-element set $\{\star\}$. Its only constant is $\star$. There are no function symbols.

Sort $\mathbf{int}$ is interpreted by the integers with infinity. Constants are the integers plus ∞. Function symbols are the usual $- : \mathbf{int} \to \mathbf{int}$ and $+, \cdot, /, \% : \mathbf{int} \times \mathbf{int} \to \mathbf{int}$ (where $/$ and $\%$ denote integer division and remainder, respectively).

Sort $\mathbf{str}$ is interpreted by the set of strings (over some fixed but unspecified alphabet). Constants are all strings. The only function symbol is $++ : \mathbf{str} \times \mathbf{str} \to \mathbf{str}$ (concatenation).

Sort $\mathbf{res}$ is interpreted by an arbitrary infinite set (whose elements are termed *resources*). There are no constants, and $\mathit{fromstr} : \mathbf{str} \to \mathbf{res}$, an embedding of strings into resources, is the only one function symbol.

Sort $\mathbf{res\{\}}$ is interpreted by multisets of resources. It features the constant $\emptyset$ (empty multiset) and the function symbols $\cap, \cup, \uplus : \mathbf{res\{\}} \times \mathbf{res\{\}} \to \mathbf{res\{\}}$ (intersection, union and sum of multisets, respectively), $|\_| : \mathbf{res\{\}} \to \mathbf{int}$ (size of a multiset), $\mathit{count} : \mathbf{res\{\}} \times \mathbf{res} \to \mathbf{int}$ (counting the multiplicity of a resource in a multiset) and $\{\_{:}\_\} : \mathbf{res} \times \mathbf{int} \to \mathbf{res\{\}}$ (constructing a "singleton" multiset containing a given resource with a given multiplicity and nothing else).

Sort $\tau[]$ is interpreted by integer-indexed arrays of elements of sort τ, where an integer-indexed array is a function from an initial segment of the natural numbers to τ. This sort features the constant null (array of length 0) and the function symbols $\mathit{len} : \tau[] \to \mathbf{int}$ (length of an array), $\_[\_] : \tau[] \times \mathbf{int} \to \tau$ (reading at a given index) and $\_[\_{:=}\_] : \tau[] \times \mathbf{int} \times \tau \to \tau[]$ (updating a given index with a given value). Note that the values of $a[i]$ and $a[i{:=}v]$ are generally unspecified if the index i is out of bounds (i. e., $i < 0$ or $i \geq \mathit{len}(a)$). As an exception, for $i = \mathit{len}(a)$, the array $a[i{:=}v]$ properly extends a, i. e., $\mathit{len}(a[i{:=}v]) = \mathit{len}(a) + 1$. This models vectors that can grow in size.

Treating the type $\mathbf{mgr}$ as an alias for the sort $\mathbf{res\{\}}$, type environments can be seen as associating sorts to variables. Given a type environment Γ and constraint $\phi \in \mathcal{L}$, we write $\Gamma \vdash \phi$ if ϕ is well-sorted w. r. t. Γ; note that this entails $\mathit{free}(\phi) \subseteq \mathit{dom}(\Gamma)$, where $\mathit{free}(\phi)$ is the set of free variables in ϕ.

Substitutions. A *substitution* μ maps variables $x \in \langle\text{var}\rangle$ to values $\mu(x) \in \langle\text{val}\rangle$ (which are variables again or constants, not arbitrary terms). We denote the domain of a substitution μ by $\mathit{dom}(\mu)$. Given a type environment Γ, we write $\Gamma\mu$ for the type environment that arises from substituting the variables in Γ according to μ. This is defined recursively: $\emptyset\mu = \emptyset$ and $(\Gamma, x{:}\tau)\mu$ equals $\Gamma\mu, x{:}\tau$ if $x \notin \mathit{dom}(\mu)$, or $\Gamma\mu, \mu(x){:}\tau$ if $\mu(x) \in \langle\text{var}\rangle$, or $\Gamma\mu$ if $\mu(x) \in \langle\text{const}\rangle$. Note that $\Gamma\mu$ need not be linear even if Γ is. Given a formula ϕ such that $\Gamma \vdash \phi$, we write $\phi\mu$ for the formula obtained by substituting the free variables of ϕ according to μ, avoiding capture. Note that $\Gamma \vdash \phi$ implies $\Gamma\mu \vdash \phi\mu$.

Valuations. Let $\mathit{\Gamma}$ be a type environment. A *$\mathit{\Gamma}$-valuation* α maps variables $x \in \mathit{dom}(\mathit{\Gamma})$ to elements $\alpha(x)$ in the interpretation of the sort $\mathit{\Gamma}(x)$; we call α a *valuation* if we do not care about the particular type environment $\mathit{\Gamma}$. We denote the domain of α by $\mathit{dom}(\alpha)$. Note that $\mathit{dom}(\alpha) \subseteq \mathit{dom}(\mathit{\Gamma})$ but not necessarily $\mathit{dom}(\alpha) = \mathit{dom}(\mathit{\Gamma})$; we call α a *maximal* $\mathit{\Gamma}$-valuation if $\mathit{dom}(\alpha) = \mathit{dom}(\mathit{\Gamma})$. Given a $\mathit{\Gamma}$-valuation α and a set of variables X, we denote the restriction of α to X by $\alpha|_X$; note that $\mathit{dom}(\alpha|_X) = \mathit{dom}(\alpha) \cap X$. Restriction induces a partial order $\succeq$ on $\mathit{\Gamma}$-valuations, where $\alpha' \succeq \alpha$ iff $\alpha'|_{\mathit{dom}(\alpha)} = \alpha$. Given n pairwise distinct variables $x_i \in \mathit{dom}(\mathit{\Gamma})$ and corresponding elements d_i in the interpretation of $\mathit{\Gamma}(x_i)$, we write $\alpha\{x_1 \mapsto d_1, \dots, x_n \mapsto d_n\}$ for the $\mathit{\Gamma}$-valuation α' that maps the x_i to d_i and all other $x \in \mathit{dom}(\alpha)$ to $\alpha(x)$. In the special case $\mathit{dom}(\alpha) = \emptyset$, we may drop α and simply write $\{x_1 \mapsto d_1, \dots, x_n \mapsto d_n\}$.

Entailment. Let $\phi, \psi \in \mathcal{L}$ be constraints such that $\mathit{\Gamma} \vdash \phi$ and $\mathit{\Gamma} \vdash \psi$. Given a $\mathit{\Gamma}$-valuation α with $\mathit{free}(\phi) \subseteq \mathit{dom}(\alpha)$, we write $\alpha \models \phi$ if α satisfies ϕ. We write $\models \phi$ if $\alpha \models \phi$ for all $\mathit{\Gamma}$-valuations α with $\mathit{free}(\phi) \subseteq \mathit{dom}(\alpha)$, and we write $\phi \models \psi$ if $\alpha \models \phi$ implies $\alpha \models \psi$ for all $\mathit{\Gamma}$-valuations α with $\mathit{free}(\phi) \cup \mathit{free}(\psi) \subseteq \mathit{dom}(\alpha)$. Entailment induces a theory $\mathcal{T} = \{\phi \mid \mathit{free}(\phi) = \emptyset \wedge \top \models \phi\}$, with respect to which entailment can be reduced to unsatisfiability. Note that unsatisfiability w. r. t. $\mathcal{T}$ is not even semi-decidable as $\mathcal{T}$ contains Peano arithmetic. Thus for reasoning purposes, we will generally approximate $\mathcal{T}$ by weaker theories.

Effects. Let f be a built-in function with $\mathit{\Pi}(f) = \mathit{\Gamma} \longrightarrow \mathit{\Delta}$ (viewing argument and return types of f as type environments $\mathit{\Gamma}$ and $\mathit{\Delta}$, respectively.) An *effect* for f is a pair of constraints ϕ and ψ such that $\mathit{\Gamma} \vdash \phi$ and $\mathit{\Gamma}, \mathit{\Delta} \vdash \psi$. (Note that $\mathit{\Gamma} \longrightarrow \mathit{\Delta}$ being a function type implies $\mathit{dom}(\mathit{\Gamma}) \cap \mathit{dom}(\mathit{\Delta}) = \emptyset$, hence $\mathit{\Gamma}, \mathit{\Delta}$ is a type environment.) We write $\phi \rightarrow \psi$ to denote such an effect, and we call ϕ its *precondition* and ψ its *action*.

An *effect environment* maps the built-in functions $f \in \mathit{dom}(\mathit{\Pi}_0)$ to effects for f. Figure 4 displays the effect environment $\mathit{\Theta}_0$, providing an axiomatic, relational semantics for all $f \in \mathit{dom}(\mathit{\Pi}_0)$. This semantics ties most built-in functions to corresponding logical operators in a straightforward way; note the non-trivial preconditions for division, reading and writing arrays, and constructing singleton multisets. The effects of functions operating on resource managers warrant some explanation.

init returns an empty manager m'.

enable non-deterministically adds some sub-multiset of r to manager m, returning the result in manager m'; the complement of the added multiset is returned in r'. In an implementation [3] the multiset to be added to m would be chosen by some *policy*, perhaps involving security profiles or user input; we use non-determinism to abstractly model such policy mechanisms.

split splits the multiset held by manager m and distributes it to the managers m'_1 and m'_2 such that m'_2 gets the largest possible sub-multiset of r.

join adds the multisets held by managers m_1 and m_2, returning their sum in m'.

consume is an explicit destructor for manager m and all its resources; the linear type system means that calls to **consume** are necessary even if m is known to be empty.

assertEmpty acts as identity on managers, but subject to the precondition that m is empty; it will be treated specially by the programming language semantics.

f	$\mathit{\Pi}_0(f)$	$\mathit{\Theta}_0(f)$
$\mathbf{id}_\tau$	$(x{:}\tau) \longrightarrow (x'{:}\tau)$	$\top \rightarrow x' \approx x$
$\mathbf{eq}_\tau$	$(x_1{:}\tau,x_2{:}\tau) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx 1 \land x_1 \approx x_2 \lor i' \approx 0 \land x_1 \not\approx x_2$
$\mathbf{add}$	$(i_1{:}\mathbf{int},i_2{:}\mathbf{int}) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx i_1 + i_2$
$\mathbf{sub}$		$\top \rightarrow i' \approx i_1 + (-i_2)$
$\mathbf{mul}$		$\top \rightarrow i' \approx i_1 \cdot i_2$
$\mathbf{div}$		$i_2 \not\approx 0 \rightarrow i' \approx i_1 \mathbin{/} i_2$
$\mathbf{mod}$		$i_2 \not\approx 0 \rightarrow i' \approx i_1 \mathbin{\%} i_2$
$\mathbf{leq}$		$\top \rightarrow i' \approx 1 \land i_1 \leq i_2 \lor i' \approx 0 \land i_1 \nleq i_2$
$\mathbf{conc}$	$(w_1{:}\mathbf{str},w_2{:}\mathbf{str}) \longrightarrow (w'{:}\mathbf{str})$	$\top \rightarrow w' \approx w_1 \mathbin{++} w_2$
$\mathbf{fromstr}$	$(w{:}\mathbf{str}) \longrightarrow (c'{:}\mathbf{res})$	$\top \rightarrow c' \approx \mathit{fromstr}(w)$
$\mathbf{null}_\tau$	$() \longrightarrow (a'{:}\tau[])$	$\top \rightarrow a' \approx \mathit{null}$
$\mathbf{length}_\tau$	$(a{:}\tau[]) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx \mathit{len}(a)$
$\mathbf{read}_\tau$	$(a{:}\tau[],i{:}\mathbf{int}) \longrightarrow (x'{:}\tau)$	$0 \leq i \land i < \mathit{len}(a) \rightarrow x' \approx a[i]$
$\mathbf{write}_\tau$	$(a{:}\tau[],i{:}\mathbf{int},x{:}\tau) \longrightarrow (a'{:}\tau[])$	$0 \leq i \land i \leq \mathit{len}(a) \rightarrow a' \approx a[i{:=}x]$
$\mathbf{empty}$	$() \longrightarrow (r'{:}\mathbf{res\{\}})$	$\top \rightarrow r' \approx \emptyset$
$\mathbf{single}$	$(c{:}\mathbf{res},i{:}\mathbf{int}) \longrightarrow (r'{:}\mathbf{res\{\}})$	$i \geq 0 \rightarrow r' \approx \{c{:}i\}$
$\mathbf{inter}$	$(r_1{:}\mathbf{res\{\}},r_2{:}\mathbf{res\{\}}) \longrightarrow (r'{:}\mathbf{res\{\}})$	$\top \rightarrow r' \approx r_1 \cap r_2$
$\mathbf{union}$		$\top \rightarrow r' \approx r_1 \cup r_2$
$\mathbf{sum}$		$\top \rightarrow r' \approx r_1 \uplus r_2$
$\mathbf{size}$	$(r{:}\mathbf{res\{\}}) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx \lvert r \rvert$
$\mathbf{count}$	$(r{:}\mathbf{res\{\}},c{:}\mathbf{res}) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx \mathit{count}(r, c)$
$\mathbf{include}$	$(r_1{:}\mathbf{res\{\}},r_2{:}\mathbf{res\{\}}) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx 1 \land r_1 \subseteq r_2 \lor i' \approx 0 \land r_1 \nsubseteq r_2$
$\mathbf{init}$	$() \longrightarrow (m'{:}\mathbf{mgr})$	$\top \rightarrow m' \approx \emptyset$
$\mathbf{enable}$	$(m{:}\mathbf{mgr},r{:}\mathbf{res\{\}}) \longrightarrow (m'{:}\mathbf{mgr},r'{:}\mathbf{res\{\}})$	$\top \rightarrow r' \subseteq r \land m \uplus r \approx m' \uplus r'$
$\mathbf{split}$	$(m{:}\mathbf{mgr},r{:}\mathbf{res\{\}}) \longrightarrow (m_1'{:}\mathbf{mgr},m_2'{:}\mathbf{mgr})$	$\top \rightarrow m_2' \approx m \cap r \land m \approx m_1' \uplus m_2'$
$\mathbf{join}$	$(m_1{:}\mathbf{mgr},m_2{:}\mathbf{mgr}) \longrightarrow (m'{:}\mathbf{mgr})$	$\top \rightarrow m' \approx m_1 \uplus m_2$
$\mathbf{consume}$	$(m{:}\mathbf{mgr}) \longrightarrow ()$	$\top \rightarrow \top$
$\mathbf{assertEmpty}$	$(m{:}\mathbf{mgr}) \longrightarrow (m'{:}\mathbf{mgr})$	$m \approx \emptyset \rightarrow m' \approx m$
$\mathbf{assertAtLeast}$	$(m{:}\mathbf{mgr},r{:}\mathbf{res\{\}}) \longrightarrow (m'{:}\mathbf{mgr})$	$r \subseteq m \rightarrow m' \approx m$

Fig. 4. Types and effects of built-in functions. The subscripts τ indicate families of functions indexed by $\tau \in \langle\text{datatype}\rangle$, except for $\mathbf{id}_\tau$, which is indexed by $\tau \in \langle\text{type}\rangle$.

$\mathbf{assertAtLeast}$ acts as identity on managers, but subject to the precondition that the manager m contains the multiset r; will be treated specially by the programming language semantics.

To facilitate the presentation of programming language semantics, we capture the logical semantics of effects directly in terms of valuations. Given a built-in function f with $\mathit{\Pi}_0(f) = \mathit{\Gamma} \longrightarrow \mathit{\Delta}$ and $\mathit{\Theta}_0(f) = \phi \rightarrow \psi$, we define $\mathit{Eff}^{\mathit{\Pi}_0}_{\mathit{\Theta}_0}(f)$ to be the set of maximal $(\mathit{\Gamma}, \mathit{\Delta})$-valuations such that $\alpha \in \mathit{Eff}^{\mathit{\Pi}_0}_{\mathit{\Theta}_0}(f)$ if and only if $\alpha \models \phi \land \psi$.

2.3 Small-step Reduction Semantics

We present a stack-based reduction semantics (which is essentially a continuation semantics) for our programming language. We will show that reduction preserves the resources stored in resource managers, thanks to linearity. Throughout this section, let $\varPi$ be a fixed well-typed program.

Stacks. We call a tuple $\langle x_1, \ldots, x_n | \alpha, e \rangle$ a *frame* if $x_1, \ldots, x_n$ is a list of pairwise distinct variables, α is a valuation and e is an expression such that

- $\mathit{dom}(\alpha) \cap \{x_1, \ldots, x_n\} = \emptyset$ and
- $\mathit{dom}(\alpha) \subseteq \mathit{free}(e) \subseteq \mathit{dom}(\alpha) \cup \{x_1, \ldots, x_n\}$.

The roles of e (redex) and α (providing values for the free variables of e) should be clear. The x_i are only present if the frame is suspended waiting for a function to return in which case the x_i act as slots for the return values. A *pre-stack* is either $\lightning$ or ϵ or $F :: S$, where F is a frame and S is a pre-stack. (Pre-stacks essentially correspond to continuations in an abstract machine interpreting λ-terms in ANF [10].) A *stack* (or *$\varPi$-stack* if we want to emphasise the program $\varPi$) is a pre-stack of the form $\lightning$ or $\langle | \alpha, e \rangle :: S$. We call $\lightning$ the *error stack*. A stack of the form $\langle | \alpha, \mathbf{ret}\ (x_1, \ldots, x_n) \rangle :: \epsilon$ is called *terminal*. If $F :: S$ is a stack then F is its *top frame*.

Reduction. Figure 5 presents the rules generating the reduction relation $\rightsquigarrow_{\varPi}$ on stacks. We denote the reflexive-transitive closure of $\rightsquigarrow_{\varPi}$ by $\rightsquigarrow_{\varPi}^{*}$. As usual $\varPi$ may be omitted if it is understood. Note that reduction performs an eager garbage collection in that it deallocates unused variables immediately by restricting the valuation α in the post stack to the free variables of the expression e.

Reduction is deterministic, except for calls to the built-in function $\mathbf{enable}$.

Proposition 2. *For all stacks S_0 there is at most one stack S_1 such that $S_0 \rightsquigarrow S_1$, unless S_0 is of the form $\langle | \alpha, \mathbf{let}\ (m', r') = \mathbf{enable}\ (m, r)\ \mathbf{in}\ e \rangle :: S_0'$.*

Typed stacks. Reduction is untyped since type information is not needed at runtime. However, various properties of reduction are best stated if the type of variables is known. Therefore, we annotate stacks with type environments and conservatively extend reduction to typed stacks.

Given a frame $\langle x_1, \ldots, x_n | \alpha, e \rangle$, we call $\langle x_1, \ldots, x_n | \alpha, e \rangle^{\varGamma}$ a *typed frame* if $\varGamma$ is a linear type environment such that

- $\mathit{dom}(\varGamma) = \mathit{dom}(\alpha) \cup \{x_1, \ldots, x_n\}$,
- α is a $\varGamma$-valuation, and
- $\varGamma \vdash e : \sigma$ for some product type σ.

A *typed pre-stack* is $\lightning$, or ϵ, or $F :: \epsilon$ where F is a typed frame, or $F :: F' :: S'$ where S' is a typed pre-stack and $F = \langle x_1, \ldots, x_m | \alpha, e \rangle^{\varGamma}$ and $F' = \langle x_1', \ldots, x_n' | \alpha', e' \rangle^{\varGamma'}$ are typed frames such that $\varGamma \vdash e : (z_1' : \varGamma'(x_1'), \ldots, z_n' : \varGamma'(x_n'))$ for some variables $z_1', \ldots, z_n'$. A *typed stack* is typed pre-stack of the form $\lightning$ or $\langle | \alpha, e \rangle^{\varGamma} :: S$. Given a typed frame $F = \langle x_1, \ldots, x_n | \alpha, e \rangle^{\varGamma}$, we denote its underlying frame $\langle x_1, \ldots, x_n | \alpha, e \rangle$ by $F^{\natural}$. We extend this notation to typed (pre-)stacks, writing $S^{\natural}$ for the (pre-)stack underlying the typed (pre-)stack S.

The following proposition shows that reduction does not break the invariants maintained by typed stacks.

$$\text{(R-ret)}\ \frac{\alpha'' = \alpha'\{x'_1 \mapsto \alpha(x_1), \ldots, x'_n \mapsto \alpha(x_n)\}}{\langle|\alpha, \mathbf{ret}\ (x_1,\ldots,x_n)\rangle :: \langle x'_1, \ldots, x'_n|\alpha', e'\rangle :: S \ \rightsquigarrow\ \langle|\alpha''|_{\mathit{free}(e')}, e'\rangle :: S}$$

$$\text{(R-let}_1^{\text{tl}}\text{)}\ \frac{\begin{array}{c}\mathit{\Pi}(f) = \boldsymbol{\lambda} e\ \mathbf{:}\ (z_1\mathbf{:}\tau_1,\ldots,z_m\mathbf{:}\tau_m) \longrightarrow \sigma' \\ \alpha' = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\}\end{array}}{\langle|\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ \mathbf{ret}\ (x'_1,\ldots,x'_n)\rangle :: S \rightsquigarrow \langle|\alpha'|_{\mathit{free}(e)}, e\rangle :: S}$$

$$\text{(R-let}_1\text{)}\ \frac{\begin{array}{c}\mathit{\Pi}(f) = \boldsymbol{\lambda} e\ \mathbf{:}\ (z_1\mathbf{:}\tau_1,\ldots,z_m\mathbf{:}\tau_m) \longrightarrow \sigma' \qquad e' \neq \mathbf{ret}\ (x'_1,\ldots,x'_n) \\ \alpha' = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\}\end{array}}{\begin{array}{c}\langle|\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e'\rangle :: S \\ \rightsquigarrow\ \langle|\alpha'|_{\mathit{free}(e)}, e\rangle :: \langle x'_1, \ldots, x'_n|\alpha|_{\mathit{free}(e')}, e'\rangle :: S\end{array}}$$

$$\text{(R-let}_2\text{)}\ \frac{\begin{array}{c}\mathit{\Pi}_0(f) = (z_1\mathbf{:}\tau_1,\ldots,z_m\mathbf{:}\tau_m) \longrightarrow (z'_1\mathbf{:}\tau'_1,\ldots,z'_n\mathbf{:}\tau'_n) \\ \alpha_f = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\} \qquad \alpha'_f \in \mathit{Eff}^{\mathit{\Pi}_0}_{\mathit{\Theta}_0}(f) \qquad \alpha'_f \succeq \alpha_f \\ \alpha' = \alpha\{x'_1 \mapsto \alpha'_f(z'_1), \ldots, x'_n \mapsto \alpha'_f(z'_n)\}\end{array}}{\langle|\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e'\rangle :: S \ \rightsquigarrow\ \langle|\alpha'|_{\mathit{free}(e')}, e'\rangle :: S}$$

$$\text{(R-let}_2^{\lightning}\text{)}\ \frac{\begin{array}{c}\mathit{\Pi}_0(f) = (z_1\mathbf{:}\tau_1,\ldots,z_m\mathbf{:}\tau_m) \longrightarrow \sigma' \qquad f \in \{\mathbf{assertEmpty}, \mathbf{assertAtLeast}\} \\ \alpha_f = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\} \qquad \forall \alpha'_f \in \mathit{Eff}^{\mathit{\Pi}_0}_{\mathit{\Theta}_0}(f) : \alpha'_f \not\succeq \alpha_f\end{array}}{\langle|\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e'\rangle :: S \ \rightsquigarrow\ \lightning}$$

$$\text{(R-if}_1\text{)}\ \frac{\alpha(v) \neq 0}{\langle|\alpha, \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\rangle :: S \ \rightsquigarrow\ \langle|\alpha|_{\mathit{free}(e_1)}, e_1\rangle :: S}$$

$$\text{(R-if}_2\text{)}\ \frac{\alpha(v) = 0}{\langle|\alpha, \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\rangle :: S \ \rightsquigarrow\ \langle|\alpha|_{\mathit{free}(e_2)}, e_2\rangle :: S}$$

Fig. 5. Small-step reduction relation $\rightsquigarrow$ (for a fixed program $\mathit{\Pi}$). Application of valuations α extends to values $v \in \langle\text{val}\rangle$ in the natural way, i. e., $\alpha(v) = v$ if v is a constant.

Proposition 3. *Let $\hat{S}_0$ be a typed stack and S_1 a stack. If $\hat{S}_0^{\natural} \rightsquigarrow S_1$ then there is a typed stack $\hat{S}_1$ such that $\hat{S}_1^{\natural} = S_1$.*

The proposition justifies the view of reduction on typed stacks as a conservative extension of the reduction relation defined in Figure 5, where reduction on typed stacks is defined by $\hat{S}_0 \rightsquigarrow_{\mathit{\Pi}} \hat{S}_1$ if and only if $\hat{S}_0^{\natural} \rightsquigarrow_{\mathit{\Pi}} \hat{S}_1^{\natural}$; as usual $\mathit{\Pi}$ may be omitted if it is understood.

We call a stack S_0 *stuck* if there is no stack S_1 such that $S_0 \rightsquigarrow S_1$, and S_0 is neither terminal nor the error stack. Our next result shows that reduction on typed stacks will get stuck only at calls to built-in functions (other than $\mathbf{assertEmpty}$ and $\mathbf{assertAtLeast}$), and only if the preconditions of these calls fail. As the effects listed in Figure 4 reveal, reduction will get stuck only upon attempts to divide by 0, access arrays out of bounds or construct singleton multisets with negative multiplicity.

Proposition 4. *Let $\hat{S}$ be a typed stack. If $\hat{S}^\natural$ is stuck then it is of the form*

$$\langle|\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e'\rangle :: S'\ ,$$

$f \in \mathit{dom}(\Pi_0) \setminus \{\mathbf{assertEmpty}, \mathbf{assertAtLeast}\}$*, and there is no* $\alpha'_f \in \mathit{Eff}^{\Pi_0}_{\Theta_0}(f)$ *such that* $\alpha'_f \succeq \alpha_f$*, where* α_f *is defined as in rule (R-let$_2$).*

Preservation of resources. Given a typed frame $F = \langle x_1, \ldots, x_n|\alpha, e\rangle^\Gamma$, we define the multiset *res*(F) of *resources* in F by *res*$(F) = \biguplus\{\alpha(x) \mid x \in \mathit{dom}(\alpha), \Gamma(x) = \mathbf{mgr}\}$. We extend *res* to typed non-error stacks by defining *res*$(\epsilon) = \emptyset$ and *res*$(F :: S) =$ *res*$(F) \uplus$ *res*(S). Proposition 5 states *resource preservation*: The sum of all resources in the system remains unchanged by reduction, unless the built-in functions $\mathbf{enable}$ and $\mathbf{consume}$ are called. The former admits increasing (but not decreasing) the resources, whereas the latter behaves the other way round. Obviously, resource preservation depends on the linearity restriction on type $\mathbf{mgr}$, otherwise resources could be duplicated by re-using managers.

Proposition 5. *Let S_0 and S_1 be typed stacks such that $S_0 \rightsquigarrow S_1 \neq \lightning$.*

1. *If S_0 is of the form $\langle|\alpha, \mathbf{let}\ (m',r') = \mathbf{enable}\ (m,r)\ \mathbf{in}\ e\rangle^\Gamma :: S'_0$ then res$(S_0) \subseteq$ res(S_1).*
2. *If S_0 is of the form $\langle|\alpha, \mathbf{let}\ () = \mathbf{consume}\ (m)\ \mathbf{in}\ e\rangle^\Gamma :: S'_0$ then res$(S_0) \supseteq$ res(S_1).*
3. *In all other cases, res$(S_0) =$ res(S_1).*

2.4 Erasing Resource Managers

According to the reduction semantics, a call to $\mathbf{assertEmpty}$ or $\mathbf{assertAtLeast}$ either does nothing[1] or goes wrong, and calling one of these two tests is the only way to go wrong. Hence, if we know that a program cannot go wrong (and Section 3 will present a type system for proving just that) then we can erase all calls to these built-ins (or rather, replace them by true no-ops) and obtain an equivalent program.

In fact, we can do more than that. Once the assertion built-ins are gone, it is even possible to remove the resource managers themselves. By the design of the programming language (in particular, the choice of built-in operations on resource managers) the contents of resource managers cannot influence the values of variables of any other type. Informally, this justifies replacing the resource managers themselves by variables of type $\mathbf{unit}$ whenever we know that a program cannot go wrong. Erasing resource managers also means that the built-in functions acting on managers can be replaced by simpler ones on $\mathbf{unit}$: all of which are no-ops, except for $\mathbf{enable}$ itself.[2] The remainder of the section formalises this intuition.

Figure 6 shows the necessary program transformations to erase resource managers. Most fundamentally, erasure maps the manager type $\mathbf{mgr}$ to the unit type $\mathbf{unit}$.

[1] Due to the linearity restriction on resource managers these functions must copy the input manager to an output manager; a true no-op would violate resource preservation.

[2] We do keep the calls in place, so that erasure preserves the structure of programs; this simplifies reasoning, and does not preclude optimising away no-op calls at a later stage.

Erasure τ° of types τ

$\tau^\circ = \mathbf{unit}$ if $\tau = \mathbf{mgr}$

$\tau^\circ = \tau$ otherwise

Erasure $\mathit{\Gamma}^\circ$ of type environments $\mathit{\Gamma}$

$\emptyset^\circ = \emptyset$

$(\mathit{\Gamma}, x{:}\tau)^\circ = \mathit{\Gamma}^\circ, x{:}\tau^\circ$

Erasure σ° of product types σ

$(x_1{:}\tau_1,\ldots,x_n{:}\tau_n)^\circ = (x_1{:}\tau_1^\circ,\ldots,x_n{:}\tau_n^\circ)$

Erasure $\mathit{\Pi}^\circ$ of programs $\mathit{\Pi}$

$dom(\mathit{\Pi}^\circ) = dom(\mathit{\Pi})$

$\mathit{\Pi}^\circ(f) = \boldsymbol{\lambda} e : \sigma^\circ \longrightarrow \sigma'^\circ$ if $\mathit{\Pi}(f) = \boldsymbol{\lambda} e : \sigma \longrightarrow \sigma'$

$\mathit{\Pi}^\circ(f) = \sigma^\circ \longrightarrow \sigma'^\circ$ if $\mathit{\Pi}(f) = \sigma \longrightarrow \sigma'$

Erasure $\mathit{\Theta}_0^\circ$ of effect environment $\mathit{\Theta}_0$

$dom(\mathit{\Theta}_0^\circ) = dom(\mathit{\Theta}_0)$

$\mathit{\Theta}_0^\circ(\mathbf{enable}) = \top \to r' \subseteq r$

$$\mathit{\Theta}_0^\circ(f) = \top \to \top \quad \text{if } \begin{cases} f \in \{\mathbf{init}, \mathbf{split}, \mathbf{join}, \mathbf{consume}\} \cup \\ \qquad \{\mathbf{assertEmpty}, \mathbf{assertAtLeast}\} \end{cases}$$

$\mathit{\Theta}_0^\circ(f) = \mathit{\Theta}_0(f)$ otherwise

Erasure α° of $\mathit{\Gamma}$-valuations α

$dom(\alpha^\circ) = dom(\alpha)$

$\alpha^\circ(x) = \star$ if $\mathit{\Gamma}(x) = \mathbf{mgr}$

$\alpha^\circ(x) = \alpha(x)$ otherwise

Erasure S° of typed stacks S

$\lightning^\circ = \lightning$ $\qquad \epsilon^\circ = \epsilon$ $\qquad (\langle x_1, \ldots, x_n | \alpha, e\rangle^{\mathit{\Gamma}} :: S)^\circ = \langle x_1, \ldots, x_n | \alpha^\circ, e\rangle^{\mathit{\Gamma}^\circ} :: S^\circ$

Fig. 6. Erasure of resource managers.

Erasure on types determines erasure on product types, type environments, programs and valuations (where erasure uniformly maps the values of $\mathbf{mgr}$-variables to $\star$, the only value of type $\mathbf{unit}$), which in turn determines erasure on typed stacks. As outlined above, erasure on effect environments trivialises the effect of resource manager built-ins, except $\mathbf{enable}$, and preserves the effects of all built-ins not operating on managers. The effect of $\mathbf{enable}$ after erasure is to non-deterministically choose a sub-multiset of r and return its complement in r'. This reflects the fact that calls to $\mathbf{enable}$ provide points of interaction for the policy (e. g., the user) to decide how many resources the system is granted. Erasing resource managers does not mean that policy decisions are fixed, it just removes the managers' book keeping about those decisions.

Lemma 6. *Let $\mathit{\Pi}$ be a well-typed program and S a typed $\mathit{\Pi}$-stack. Then $\mathit{\Pi}^\circ$ is a well-typed program and S° a typed $\mathit{\Pi}^\circ$-stack.*

Erasure makes trivial the effects of $\mathbf{assertEmpty}$ and $\mathbf{assertAtLeast}$, and in particular, replaces their precondition by $\top$. Thus a program cannot go wrong after erasure, as rule (R-let$_2^{\lightning}$) will never apply.

Proposition 7. *Let Π be a well-typed program and S a Π°-stack S. Then $S \not\rightsquigarrow^*_{\Pi^\circ} \lightning$.*

The next result states that the small-step reduction relation $\rightsquigarrow_\Pi$ of a program Π is almost bisimulation equivalent to the reduction relation $\rightsquigarrow_{\Pi^\circ}$ of its erasure. In fact, it shows that the relation $R = \{\langle S, S^\circ \rangle \mid S \text{ is a } \Pi\text{-stack}\}$ would be a bisimulation if $\rightsquigarrow_\Pi$ could not reduce stacks to the error stack $\lightning$. Put differently, if Π cannot go wrong then $\rightsquigarrow_\Pi$ and $\rightsquigarrow_{\Pi^\circ}$ are bisimulation equivalent. The proof of this theorem is by case analysis on the reduction relation $\rightsquigarrow_\Pi$ of the unerased program. As a corollary, we get that reachability in the erased program is essentially the same as reachability in the unerased one, provided that the unerased program cannot go wrong.

Theorem 8. *Let Π be a well-typed program and $\hat{S}_0$ a typed Π-stack with $\hat{S}_0 \not\rightsquigarrow_\Pi \lightning$.*

1. *For all typed Π-stacks $\hat{S}_1$, if $\hat{S}_0 \rightsquigarrow_\Pi \hat{S}_1$ then $\hat{S}_0^\circ \rightsquigarrow_{\Pi^\circ} \hat{S}_1^\circ$.*
2. *For all typed Π°-stacks S_1, if $\hat{S}_0^\circ \rightsquigarrow_{\Pi^\circ} S_1$ then there is a typed Π-stack $\hat{S}_1$ such that $\hat{S}_0 \rightsquigarrow_\Pi \hat{S}_1$ and $\hat{S}_1^\circ = S_1$.*

Corollary 9. *Let Π be a well-typed program and S_0 a typed Π-stack. If $S_0 \not\rightsquigarrow^*_\Pi \lightning$ then $\{S^\circ \mid S_0 \rightsquigarrow^*_\Pi S\} = \{S \mid S_0^\circ \rightsquigarrow^*_{\Pi^\circ} S\}$.*

What distinguishes erasure of resource managers from other erasure results (e. g., type erasure during compilation, Java generics erasure) is that here, erasure does not completely remove a language construct. Instead, it removes the book keeping but retains the semantically important bit that deals with dynamic policy decisions.

2.5 Big-step Relational Semantics

The reduction semantics presented in Section 2.3 is good for showing preservation properties, like the preservation of resources. However, it does not easily yield a relational view on functions, relating input and output parameters. This is achieved by a relational semantics, which we will prove equivalent to the reduction semantics. Contrary to the reduction semantics, which was originally untyped and had type environments added conservatively, the relational semantics will be typed from the start. (Types do not hurt here, as the relational semantics is not geared towards execution.)

Throughout this section, we assume that Π is a well-typed program. A *state* β is either the error state $\lightning$ or a normal state $\langle \Gamma; \alpha \rangle$, where Γ is a linear type environment and α a maximal Γ-valuation. Given an expression e, a normal state $\langle \Gamma; \alpha \rangle$ and a state β', we define the judgement $e, \langle \Gamma; \alpha \rangle \Downarrow_\Pi \beta'$ (or $e, \langle \Gamma; \alpha \rangle \Downarrow \beta'$ if Π is understood) by the rules in Figure 7 if $\mathit{dom}(\Gamma) \cap \mathit{bound}(e) = \emptyset$ and there are Γ_e and σ such that $\Gamma \succeq \Gamma_e$ and $\Gamma_e \vdash e : \sigma$. The intended meaning of $e, \langle \Gamma; \alpha \rangle \Downarrow \beta'$ is that evaluating expression e in state $\langle \Gamma; \alpha \rangle$ may terminate and result in state β'.

The reduction semantics deallocates variables once they become unused (an eager garbage collection, so to say), which is essential for the linear variables as otherwise resource preservation would not hold. However, the intermediate values of variables are thus lost. In contrast, the relational semantics names and records all intermediate values, even the linear ones, as $e, \langle \Gamma; \alpha \rangle \Downarrow \langle \Gamma'; \alpha' \rangle$ implies $\Gamma' \succeq \Gamma$ and $\alpha' \succeq \alpha$.

By definition, violations of resource safety manifest themselves in reductions ending in the error stack, and hence reductions which diverge or get stuck cannot

Evaluation of expressions $e, \langle \mathit{\Gamma}; \alpha \rangle \Downarrow \beta'$

$$\text{(E-ret)}\ \frac{}{\mathbf{ret}\ (x_1,\ldots,x_n), \langle \mathit{\Gamma}; \alpha \rangle \Downarrow \langle \mathit{\Gamma}; \alpha \rangle}$$

$$\text{(E-let}_1\text{)}\ \frac{\begin{array}{c} \mathit{\Pi}(f) = \boldsymbol{\lambda} e : (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \longrightarrow (z'_1{:}\tau'_1,\ldots,z'_n{:}\tau'_n) \qquad \mathit{\Gamma}_f = z_1{:}\tau_1, \ldots, z_m{:}\tau_m \\ \alpha_f = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\} \qquad e, \langle \mathit{\Gamma}_f; \alpha_f \rangle \Downarrow \langle \mathit{\Gamma}'_f; \alpha'_f \rangle \\ \mathit{\Gamma}' = \mathit{\Gamma}, x'_1{:}\tau'_1, \ldots, x'_n{:}\tau'_n \qquad \alpha' = \alpha\{x'_1 \mapsto \alpha'_f(z'_1), \ldots, x'_n \mapsto \alpha'_f(z'_n)\} \\ e', \langle \mathit{\Gamma}'; \alpha' \rangle \Downarrow \beta'' \end{array}}{\mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e', \langle \mathit{\Gamma}; \alpha \rangle \Downarrow \beta''}$$

$$\text{(E-let}_1^{\lightning}\text{)}\ \frac{\begin{array}{c} \mathit{\Pi}(f) = \boldsymbol{\lambda} e : (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \longrightarrow \sigma' \qquad \mathit{\Gamma}_f = z_1{:}\tau_1, \ldots, z_m{:}\tau_m \\ \alpha_f = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\} \qquad e, \langle \mathit{\Gamma}_f; \alpha_f \rangle \Downarrow \lightning \end{array}}{\mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e', \langle \mathit{\Gamma}; \alpha \rangle \Downarrow \lightning}$$

$$\text{(E-let}_2\text{)}\ \frac{\begin{array}{c} \mathit{\Pi}(f) = (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \longrightarrow (z'_1{:}\tau'_1,\ldots,z'_n{:}\tau'_n) \\ \alpha_f = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\} \qquad \alpha'_f \in \mathit{Eff}_{\mathit{\Theta}_0}^{\mathit{\Pi}_0}(f) \qquad \alpha'_f \succeq \alpha_f \\ \mathit{\Gamma}' = \mathit{\Gamma}, x'_1{:}\tau'_1, \ldots, x'_n{:}\tau'_n \qquad \alpha' = \alpha\{x'_1 \mapsto \alpha'_f(z'_1), \ldots, x'_n \mapsto \alpha'_f(z'_n)\} \\ e', \langle \mathit{\Gamma}'; \alpha' \rangle \Downarrow \beta'' \end{array}}{\mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e', \langle \mathit{\Gamma}; \alpha \rangle \Downarrow \beta''}$$

$$\text{(E-let}_2^{\lightning}\text{)}\ \frac{\begin{array}{c} \mathit{\Pi}(f) = \boldsymbol{\lambda} e : (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \longrightarrow \sigma' \qquad f \in \{\mathbf{assertEmpty}, \mathbf{assertAtLeast}\} \\ \alpha_f = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\} \qquad \forall \alpha'_f \in \mathit{Eff}_{\mathit{\Theta}_0}^{\mathit{\Pi}_0}(f) : \alpha'_f \not\succeq \alpha_f \end{array}}{\mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e', \langle \mathit{\Gamma}; \alpha \rangle \Downarrow \lightning}$$

$$\text{(E-if}_1\text{)}\ \frac{e_1, \langle \mathit{\Gamma}; \alpha \rangle \Downarrow \beta'}{\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2, \langle \mathit{\Gamma}; \alpha \rangle \Downarrow \beta'}\ \text{if } \alpha(v) \neq 0$$

$$\text{(E-if}_2\text{)}\ \frac{e_2, \langle \mathit{\Gamma}; \alpha \rangle \Downarrow \beta'}{\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2, \langle \mathit{\Gamma}; \alpha \rangle \Downarrow \beta'}\ \text{if } \alpha(v) = 0$$

Fig. 7. Big-step evaluation relation (for a fixed program $\mathit{\Pi}$).

violate resource safety. Therefore, resource safety is not affected by the fact that the relational semantics ignores such reductions. Under this proviso, Proposition 10 shows the equivalence of reduction and relational semantics.

Proposition 10. *Let $\langle \mathit{\Gamma}; \alpha \rangle$ and $\langle \mathit{\Gamma}'; \alpha' \rangle$ be states. Let e be an expression such that $\mathit{dom}(\mathit{\Gamma}) = \mathit{free}(e)$ and $\mathit{\Gamma} \vdash e : \sigma$ for some product type σ. Then*

1. *$e, \langle \mathit{\Gamma}; \alpha \rangle \Downarrow \lightning$ if and only if $\langle\!| \alpha, e \rangle^{\mathit{\Gamma}} :: \epsilon \leadsto^* \lightning$, and*
2. *$e, \langle \mathit{\Gamma}; \alpha \rangle \Downarrow \langle \mathit{\Gamma}'; \alpha' \rangle$ if and only if there is a typed stack $\langle\!| \alpha'', \mathbf{ret}\ (x_1,\ldots,x_n) \rangle^{\mathit{\Gamma}''} :: \epsilon$ such that $\langle\!| \alpha, e \rangle^{\mathit{\Gamma}} :: \epsilon \leadsto^* \langle\!| \alpha'', \mathbf{ret}\ (x_1,\ldots,x_n) \rangle^{\mathit{\Gamma}''} :: \epsilon$ and $\mathit{\Gamma}' \succeq \mathit{\Gamma}''$ and $\alpha' \succeq \alpha''$.*

3 Effect Type System

In this section, we will develop a type system to statically guarantee dynamic resource safety, i. e., the absence of reductions to the error stack $\lightning$. We will do so by annotating functions with effects and then extending the notion of effect to a judgement on expressions, which we will define by a simple set of typing rules.

3.1 Effect Type System

We extend the notion of effect $\phi \to \psi$ from built-in functions to λ-abstractions. To be precise, $\phi \to \psi$ is an *effect* for f if $\Gamma \vdash \phi$ and $\Gamma, \Delta \vdash \psi$, where $\Pi(f) = [\boldsymbol{\lambda} \dots]\Gamma \longrightarrow \Delta$, regardless of whether f is built-in or a λ-abstraction. In line with this extension, an *effect environment* Θ maps all functions $f \in dom(\Pi)$ to effects $\Theta(f)$ for f.

In order to derive the effects of λ-abstractions, we generalise effects to effect types for expressions and develop a type system for inductively constructing such effect types. Effects relate input and output parameters of functions by logical formulae. Likewise, effect types shall relate input and output parameters of expressions. Here, the input parameters of an expression are its free variables; the output parameters are those variables that are not free yet but will become free during reduction, i. e., the (let-)bound variables. Formally, an *effect type* $\Gamma; \phi \to \Delta; \psi$ is a pair of constraints ϕ and ψ together with a pair of type environments Γ and Δ such that $dom(\Gamma) \cap dom(\Delta) = \emptyset$ and $\Gamma \vdash \phi$ and $\Gamma, \Delta \vdash \psi$. We call ϕ and ψ *precondition* and *action*, and Γ and Δ *input* and *output (parameters)*, respectively. Given an expression e, we say that an effect type $\Gamma; \phi \to \Delta; \psi$ is an *effect type for* e if $dom(\Gamma) \cap bound(e) = \emptyset$.

We say that an effect type $\Gamma; \phi \to \Delta; \psi$ is *stronger than* an effect type $\Gamma'; \phi' \to \Delta'; \psi'$, denoted by $\Gamma; \phi \to \Delta; \psi \supseteq \Gamma'; \phi' \to \Delta'; \psi'$, if $\phi' \models \phi$ and $(\phi' \wedge \psi) \models \psi'$, i. e., the stronger effect type $\Gamma; \phi \to \Delta; \psi$ has a weaker precondition but stronger action. The stronger-than relation $\supseteq$ is a quasi-order, i. e., reflexive and transitive, and induces an equivalence relation on effect types, the *as-strong-as* relation, which we denote by $\equiv$. Note that for every effect type $\Gamma; \phi \to \Delta; \psi$ is as strong as an effect type $\Gamma'; \phi \to \Delta'; \psi$ with linear type environments Γ' and Δ'.

Figure 8 presents the typing rules for deriving effect types. There, the judgement $\Theta \vdash_\Pi e : \Gamma; \phi \to \Delta; \psi$ states that expression e has effect type $\Gamma; \phi \to \Delta; \psi$ in the context of program Π and effect environment Θ. If Π is understood, we may omit it and write $\Theta \vdash e : \Gamma; \phi \to \Delta; \psi$ instead. The judgement $\Pi, \Theta \vdash f$ means that the effect type ascribed to a λ-abstraction f by Θ and Π is consistent with the effect type derived for the body of f. We say that Θ is an *admissible* effect environment for a program Π if $\Pi, \Theta \vdash f$ for all λ-abstractions $f \in dom(\Pi) \setminus dom(\Pi_0)$.

Lemma 11. *Let e be an expression, Θ an effect environment (referring to an implicit program Π) and $\Gamma; \phi \to \Delta; \psi$ an effect type. If $\Theta \vdash e : \Gamma; \phi \to \Delta; \psi$ then $\Gamma; \phi \to \Delta; \psi$ is an effect type for e.*

Theorem 12 states soundness of effect typing w. r. t. the big-step relational semantics. The proof is by double induction on the derivation of relational semantics judgements over the derivation of effect type judgements. As a corollary, we get that reduction starting from a state that satisfies the precondition can't go wrong, hence resource managers can be erased. In fact, the untyped reductions in the erased program match exactly the typed reductions in the original program.

Typing of expression effects $\Theta \vdash e : \Gamma; \phi \to \Delta; \psi$

$$\text{(ET-weak)}\ \frac{\Theta \vdash e : \Gamma; \phi \to \Delta; \psi}{\Theta \vdash e : \Gamma'; \phi' \to \Delta'; \psi'} \ \text{if} \begin{cases} \mathit{dom}(\Gamma') \cap \mathit{bound}(e) = \emptyset \wedge \\ \Gamma; \phi \to \Delta; \psi \supseteq \Gamma'; \phi' \to \Delta'; \psi' \end{cases}$$

$$\text{(ET-ret)}\ \frac{}{\Theta \vdash \mathbf{ret}\ (x_1,\ldots,x_n) : \emptyset; \top \to \emptyset; \top}$$

$$\text{(ET-if)}\ \frac{\Theta \vdash e_1 : \Gamma; v \not\approx 0 \wedge \phi \to \Delta; \psi \qquad \Theta \vdash e_2 : \Gamma; v \approx 0 \wedge \phi \to \Delta; \psi}{\Theta \vdash \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \Gamma; \phi \to \Delta; \psi}$$

$$\text{(ET-let)}\ \frac{\begin{array}{c} \Pi(f) = [\boldsymbol{\lambda}\ldots]\Gamma \to \Delta \qquad \Gamma = z_1{:}\tau_1, \ldots, z_m{:}\tau_m \qquad \Delta = z'_1{:}\tau'_1, \ldots, z'_n{:}\tau'_n \\ \Theta(f) = \phi \to \psi \qquad \mu = \{z_1 \mapsto v_1, \ldots, z_m \mapsto v_m, z'_1 \mapsto x'_1, \ldots, z'_n \mapsto x'_n\} \\ \Theta \vdash e' : \Gamma', \Delta'; \phi' \wedge \psi' \to \Delta''; \psi'' \end{array}}{\Theta \vdash \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e' : \Gamma'; \phi' \to \Delta', \Delta''; \psi' \wedge \psi''} \ \text{if } (*)$$

$$\text{where } (*) \begin{cases} \mathit{dom}(\Gamma') \cap \{x'_1, \ldots, x'_n\} = \emptyset \wedge \\ \Gamma\mu; \phi\mu \to \Delta\mu; \psi\mu \supseteq \Gamma'; \phi' \to \Delta'; \psi' \end{cases}$$

Well-typedness of λ-abstraction effects $\Pi, \Theta \vdash f$

$$\text{(ET-lam)}\ \frac{\Pi(f) = \boldsymbol{\lambda} e : \Gamma \to \Delta \qquad \Theta(f) = \phi \to \psi \qquad \Theta \vdash e : \Gamma; \phi \to \Delta; \psi}{\Pi, \Theta \vdash f}$$

Fig. 8. Typing rules for effect types (for a fixed program Π).

Theorem 12. *Let Θ be an admissible effect environment for a well-typed program Π. Let e be an expression and $\Gamma; \phi \to \Delta; \psi$ an effect type such that $\Theta \vdash \mathrm{e} : \Gamma; \phi \to \Delta; \psi$. Let $\langle \Gamma; \alpha \rangle$ and β' be states such that $\mathrm{e}, \langle \Gamma; \alpha \rangle \Downarrow \beta'$ (which implies $\Gamma_{\mathrm{e}} \vdash \mathrm{e} : \sigma$ for some Γ_{e}, σ). If $\alpha \models \phi$ then $\beta' = \langle \Gamma'; \alpha' \rangle$ for some Γ' and α' such that $\alpha' \models \phi \wedge \psi$. (In particular, if $\alpha \models \phi$ then $\beta' \neq \lightning$.)*

Corollary 13. *Let Θ be an admissible effect environment for a well-typed program Π. Let e be an expression and $\Gamma; \phi \to \Delta; \psi$ an effect type such that $\Theta \vdash_\Pi \mathrm{e} : \Gamma; \phi \to \Delta; \psi$. Let α be a maximal Γ-valuation, and let $\hat{S}_0 = \langle |\alpha|_{\mathit{free}(\mathrm{e})}, \mathrm{e} \rangle^{\Gamma|_{\mathit{free}(\mathrm{e})}} :: \epsilon$ be a typed Π-stack (which implies $\Gamma|_{\mathit{free}(\mathrm{e})} \vdash_\Pi \mathrm{e} : \sigma$ for some σ). If $\alpha \models \phi$ then*

1. *$\hat{S}_0 \not\rightsquigarrow^*_\Pi \lightning$ and*
2. *for all (untyped) Π°-stacks S, $\hat{S}_0^{\circ\natural} \rightsquigarrow^*_{\Pi^\circ} S$ if and only if there is a typed Π-stack $\hat{S}$ such that $\hat{S}_0 \rightsquigarrow^*_\Pi \hat{S}$ and $\hat{S}^{\circ\natural} = S$. (In particular, $\hat{S}_0^{\circ\natural} \not\rightsquigarrow^*_{\Pi^\circ} \lightning$.)*

3.2 Example: Bulk Messaging Application

To illustrate the use of the effect type system, we revisit the example from Figure 1. The interesting bits of code are in the functions send_bulk and send_msg.

The function send_bulk first builds up a multiset of resources r by converting the strings representing phone numbers in nums into resources. Next it attempts to authorise the use of all resources by having enable add r to an empty resource manager m. If this

f	$\Theta(f)$
send_bulk	$\top \rightarrow \top$
res_from_nums	$\top \rightarrow \mathsf{r} \approx \mathit{bagof}(\mathit{map}_{\mathit{fromstr}}(\mathsf{nums}))$
res_from_nums'	$0 \leq \mathsf{i} \leq \mathit{len}(\mathsf{nums}) \wedge \mathsf{r'} \approx \mathit{bagof}(\mathit{map}_{\mathit{fromstr}}(\mathit{subarray}(\mathsf{nums}, \mathsf{i}, \mathit{len}(\mathsf{nums}))))$ $\rightarrow \mathsf{r} \approx \mathit{bagof}(\mathit{map}_{\mathit{fromstr}}(\mathsf{nums}))$
send_msgs	$\mathit{bagof}(\mathit{map}_{\mathit{fromstr}}(\mathsf{nums})) \subseteq \mathsf{m} \rightarrow \mathsf{m} \approx \mathsf{m'} \uplus \mathit{bagof}(\mathit{map}_{\mathit{fromstr}}(\mathsf{nums}))$
send_msgs'	$0 \leq \mathsf{i} \leq \mathit{len}(\mathsf{nums}) \wedge \mathit{bagof}(\mathit{map}_{\mathit{fromstr}}(\mathit{subarray}(\mathsf{nums}, 0, \mathsf{i}))) \subseteq \mathsf{m}$ $\rightarrow \mathsf{m} \approx \mathsf{m'} \uplus \mathit{bagof}(\mathit{map}_{\mathit{fromstr}}(\mathit{subarray}(\mathsf{nums}, 0, \mathsf{i})))$
send_msg	$\mathit{count}(\mathsf{m}, \mathit{fromstr}(\mathsf{num})) \geq 1 \rightarrow \mathsf{m} \approx \mathsf{m'} \uplus \{\!\!\{\mathit{fromstr}(\mathsf{num}){:}1\}\!\!\}$
prim_send_msg	$\top \rightarrow \top$

$$\forall a : \mathit{len}(\mathit{map}_{\mathit{fromstr}}(a)) \approx \mathit{len}(a)$$
$$\forall a \forall i : 0 \leq i < \mathit{len}(a) \Rightarrow \mathit{map}_{\mathit{fromstr}}(a)[i] \approx \mathit{fromstr}(a[i])$$

$$\forall a \forall j \forall k : 0 \leq j \leq k \leq \mathit{len}(a) \Rightarrow \mathit{len}(\mathit{subarray}(a, j, k)) = k + (-j)$$
$$\forall a \forall j \forall k \forall i : 0 \leq j \leq k \leq \mathit{len}(a) \wedge 0 \leq i < \mathit{len}(\mathit{subarray}(a, j, k)) \Rightarrow \mathit{subarray}(a, j, k)[i] = a[j + i]$$

$$\forall a : |\mathit{bagof}(a)| \approx \mathit{len}(a)$$
$$\forall a : \mathit{len}(a) \approx 1 \Rightarrow \mathit{bagof}(a) \approx \{a[0]{:}1\}$$
$$\forall a \forall k : 0 \leq k \leq \mathit{len}(a) \Rightarrow \mathit{bagof}(a) \approx \mathit{bagof}(\mathit{subarray}(a, 0, k)) \uplus \mathit{bagof}(\mathit{subarray}(a, k, \mathit{len}(a)))$$

Fig. 9. Bulk messaging application: admissible effect environment Θ and axiomatisation of theory extension; for the sake of readability sort information is suppressed in the axioms.

fails, i. e., the multiset r' returned by enable is of non-zero size, send_bulk terminates (after destroying m' and whatever resources it holds).[3] If authorising all resources succeeds, send_bulk calls send_msgs to actually send the messages while checking that the manager m' contains the required resources. After that, send_bulk checks that send_msgs has used up all resources by asserting that the returned manager m'' is empty; failing this assertion will trigger a runtime error. Finally, send_bulk explicitly destroys the empty manager m''' and terminates.

The function send_msg sends one message, checking whether the resource manager m holds the resource required. It does so by converting the string num into a singleton multiset of resources r. Then it splits the manager m into m' and m_r, so that m_r contains at most the resources in r. Next, send_msg asserts that m_r contains at least r; failing this assertion will trigger a runtime error. Succeeding the assertion, send_msg calls the primitive send function, destroys the now used resource by consuming m_r', and returns the remaining resources in the manager m'.

The bulk messaging example is statically resource safe, as witnessed by the admissible effect environment displayed in Figure 9. Of particular interest is the effect $\top \rightarrow \top$ ascribed to the main function send_bulk. This least informative effect expresses nothing about the function itself but implies the absence of runtime errors via Corollary 13.

The effects require an extension of the theory $\mathcal{T}$ (see Section 2.2) by three new functions, axiomatised in Figure 9. The function *map* maps an array of strings to an

[3] A more sophisticated version of the application could deal more gracefully with enable granting only part of the requested resources. This would require more complex code to inspect the multisets r and r' (but not the resource manager m').

array of resources, $subarray$ takes an array and cuts out the sub-array between two given indices, and $bagof$ converts an array of resources to a multiset (containing the same elements with the same multiplicity). Note that the axiomatisation of $bagof$ is not complete[4] but sufficient for our purposes.

Effect type checking, e. g., for checking admissibility of the effect environment Θ from Figure 9, requires checking the side condition of the weakening rule (ET-weak), which involves checking logical entailment w. r. t. to an extension of the theory $\mathcal{T}$. Due to the high undecidability of $\mathcal{T}$, we actually check entailment w. r. t. (an extension of) an approximation of $\mathcal{T}$; in particular, we approximate multiplication and division by uninterpreted functions. For the bulk messaging example, we used an SMT solver [4] that can handle linear integer arithmetic and arrays. We added axioms for multisets and the axioms in Figure 9. Due to an incomplete quantifier instantiation heuristic, we had to instantiate a number of these axioms by hand, yet eventually, the solver was able to prove all the entailments required by the weakening rules.

Even though arising from a single example, we believe that the extension of the theories of multisets and arrays with the functions $subarray$ and $bagof$ is quite generic and could prove useful in many cases.

4 Conclusion

We have presented a programming language with support for complex resource management, close to the standard SSA/ANF forms of compiler intermediate languages [1]. By construction, programs are *dynamically resource safe* in that any attempts to abuse resources are trapped. We have extended the language with an effect type system which guarantees the for well-typed programs no such attempts occur: we have *static resource safety*. In addition, for such programs the bookkeeping required by dynamic resource management can be erased.

Related Work. Many tools and methods have been proposed to assist with resource management at runtime, e.g., in Java, the JRes [9] and J-Seal [8] frameworks. Generally, these aim to enable programs to react to fluctuations of resources caused by an unpredictable environment. Our aim, however is to track the flow of resources through the program, where the environment can influence the availability of resources only at well-understood points of interaction with the program and with clear availability policies. This offers the chance for more precise resource control whose behaviour can be predicted statically.

This paper builds on previous work [3] with a Java library implementing resource managers and focusing on the dynamic aspects of resource management policies. This Java library supports essentially the same operations on resource managers as our functional language, except that state is realised by destructive updates instead of linear types. While [3] does not provide a static analysis to prove static resource safety, it does outline how dynamic accounting could be erased if static resource safety were provable. Our work here shows one way to do just that.

[4] A complete axiomatisation of $bagof$ is possible in the full first-order theory of multisets and arrays but it is much more complicated and unusable in practise.

Our approach is in line with a general trend of providing the programmer with language-based mechanisms for security and additional static analyses (often using type systems) which use these mechanisms. This combination provides a desirable graceful degradation: if static analysis succeeds in proving certain properties, then the program may be optimised without affecting security. Yet, even if the analyses fail the language based mechanisms will enforce the security properties at runtime.

The context of our work is the MOBIUS project [5] on proof-carrying code (PCC) for mobile devices. Our effect type system is very simple and in principle well-suited for a PCC setting where checkers themselves are resource bounded. However, the weakening rule relies on checking logical entailment in a first-order theory, which is undecidable in general. Therefore, a certificate for PCC need not only provide a type derivation tree but also proofs (in some proof system) for the entailment checks in the weakening rule. The development of a suitable such proof system is a topic for further research, as is the investigation of decidable fragments of relevant first-order theories.

Acknowledgements. This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905. This paper reflects only the authors' views and the European Community is not liable for any use that may be made of the information contained therein. Ian Stark was also supported by an Advanced Research Fellowship from the UK Engineering and Physical Sciences Research Council, EPSRC project GR/R76950/01.

References

[1] A. W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[2] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resources. *Theoret. Comput. Sci.*, 389(3):411–445, 2007.

[3] D. Aspinall, P. Maier, and I. Stark. Monitoring external resources in Java MIDP. *Electron. Notes Theor. Comput. Sci.*, 197:17–30, 2008.

[4] C. Barrett, L. de Moura, and A. Stump. Design and results of the 2nd annual satisfiability modulo theories competition. *Form. Meth. Syst. Des.*, 31(3):221–239, 2007.

[5] G. Barthe, L. Beringer, P. Crégut, B. Grégoire, M. Hofmann, P. Müller, E. Poll, G. Puebla, I. Stark, and E. Vétillard. MOBIUS: Mobility, ubiquity, security. Objectives and progress report. In *Proc. TGC 2006*, LNCS 4661, pp.10–29. Springer, 2007.

[6] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic certification of heap consumption. In *Proc. LPAR 2004*, LNCS 3452, pp.347–362. Springer, 2005.

[7] F. Besson, G. Dufay, and T. P. Jensen. A formal model of access control for mobile interactive devices. In *Proc. ESORICS 2006*, LNCS 4189, pp.110–126. Springer, 2006.

[8] W. Binder, J. Hulaas, and A. Villazón. Portable resource control in Java. In *Proc. OOPSLA 2001*, pp.139–155. ACM, 2001.

[9] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proc. OOPSLA '98*, pp.21–35. ACM, 1998.

[10] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. PLDI '93*, pp.237–247. ACM, 1993.

[11] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *Proc. ESOP 2007*, LNCS 4421, pp.189–204. Springer, 2007.

[12] Unknown. Redbrowser.A, Feb. 2006. J2ME trojan, variously identified in the wild as *Redbrowser.A* (F-Secure), *J2ME/Redbrowser.a* (McAfee), *Trojan.Redbrowser.A* (Symantec), *Trojan-SMS.J2ME.Redbrowser.a* (Kaspersky Lab).