Resource Bound Certification*

Karl Crary
Carnegie Mellon University

Stephanie Weirich
Cornell University

Abstract

Various code certification systems allow the certification and static verification of important safety properties such as memory and control-flow safety. These systems are valuable tools for verifying that untrusted and potentially malicious code is safe before execution. However, one important safety property that is not usually included is that programs adhere to specific bounds on resource consumption, such as running time.

We present a decidable type system capable of specifying and certifying bounds on resource consumption. Our system makes two advances over previous resource bound certification systems, both of which are necessary for a practical system: We allow the execution time of programs and their subroutines to vary, depending on their arguments, and we provide a fully automatic compiler generating certified executables from source-level programs. The principal device in our approach is a strategy for simulating dependent types using sum and inductive kinds.

1 Introduction

A current trend in systems software is to allow untrusted extensions to be installed in protected services, relying on language technology to protect the integrity of the service instead of hardware-based protection mechanisms [11, 20, 1, 16, 14]. For example, the SPIN project [1] relies on the Modula-3 type system to protect an operating system kernel from erroneous extensions. Similarly, web browsers rely on the Java Virtual Machine bytecode verifier [11] to protect users from malicious applets. In both situations, the goal is to eliminate expensive inter-process communications or boundary crossings by allowing extensions to access directly the resources they require.

Recently, Necula and Lee [16, 15] have proposed Proof-Carrying Code (PCC) and Morrisett *et al.* [14] have proposed Typed Assembly Language (TAL) as language technologies that provide the security advantages of high-level languages, but without the overheads of interpretation or just-in-time compilation. In both systems, low-level machine code can be heavily optimized, by hand or by compiler, and yet be automatically verified through proof- or type-checking.

*This material is based on work supported in part by ARPA grant F19628-95-C-0050, AF grant F49620-97-1-0013, and NSF grant EIA 97-03470. The second author was supported in part by an NSF graduate fellowship.

To appear in the Twenty-Seventh Symposium on Principles of Programming Languages, Boston, Massachusetts, January 2000. Last revised November 3, 1999.

Each of these systems (SPIN, Java, TAL, and Touchstone [18], a compiler that generates PCC) automatically certifies a large set of important security properties, such as type safety, memory safety, and control-flow safety. However, one important security property that none of these systems certifies is bounded termination; none can guarantee that an accepted program will terminate within a given amount of time. Such guarantees of running-time bounds (and, more generally, of resource-consumption bounds) are essential to many applications, such as active networks and extensible operating system kernels. To obtain such bounds on resource consumption, code consumers have generally had to rely on operating system monitoring, which can be costly and which works against the direct access afforded by language-based mechanisms.

To redress this shortcoming, we present a decidable type system called LXres for specifying bounds on resource consumption and for certifying that programs satisfy those bounds. In this paper we will focus on specification and certification of running-time bounds; we consider at the end how our mechanisms can be generalized to other sorts of resources, such as space. Note that we make no effort to *infer* any bounds on resource consumption; rather we provide a mechanism for certifying those bounds once they have been discovered by the programmer or by program analysis.

We have implemented our type system within the framework of the Typed Assembly Language security infrastructure [14, 13]. The implemented version of this work (called TALres) lowers the mechanisms we discuss in this paper down to the TAL level, augmenting TAL's very low-level safety certification with running-time guarantees. We have also implemented a prototype certifying compiler that generates TALres, taking as its input an impure functional language (called PopCron) derived from a safe dialect of C [13] and annotated with termination information. Many of the low-level details that are the focus of TAL are not germane to certification of resource bounds, so in the interest of clarity we present our results at a higher level before discussing TALres informally.

2 Informal account

At its core, our approach is quite simple. Following Necula and Lee [17], we augment the underlying computing machinery with a *virtual clock* that winds down as programs execute. The virtual clock exists only in support of the formal semantics; the machine provides no run-time operations that can inspect the clock, and therefore the clock need not be (and is not) implemented at run time. The pertinence of the clock is to the type system: Function types, in addition to specifying the types of a function's argument and result, also specify that function's beginning and ending clock readings. For example, a function that takes an integer at time

15 and returns another integer at time 12 could be given the type:

$$(\mathtt{int}, 15) \to (\mathtt{int}, 12)$$

However, such a type is unduly specific. By specifying the starting clock to be 15, this type restricts its members so that they may be started only at one particular time. We would prefer a more flexible type the specifies that the function runs in 3 clock steps. Indeed, since there exist no operations that can inspect the clock, that is the entire import of the above type anyway.

We can get the desired type by using polymorphism to abstract over the finishing time. For example, the type

$$\forall n.\, (\mathtt{int}, n+3) \to (\mathtt{int}, n)$$

specifies that for any time n, when presented with an integer at time $n+3$, its members return an integer at time n. In other words, its members run in 3 clock steps. The former type can be obtained by instantiating n with 12, but such functions can also be run at any other time (provided the clock reads at least 3) by a suitable instantiation.

As a minor point, we also allow programs to waste clock steps on demand, so a function with the latter type could in fact take 1 or 2 steps and then waste the remaining steps as necessary to meet the specification. Also, due to some practical considerations that we discuss in Section 4, we only step the clock for function calls. However, this decision is only to simplify the compiler, and has no significant impact on the design of the language; one could just as easily use a tighter measure of running time such as the number of instructions executed.

To make this concrete, consider the function map_pair, a higher-order function that takes a function f and a pair p, and applies f to both components of p:

$$\begin{array}{l}\Lambda\alpha{:}\mathsf{Type}, \beta{:}\mathsf{Type}, k{:}\mathsf{Nat}, n{:}\mathsf{Nat}.\\ \quad \lambda(fp : (\forall m{:}\mathsf{Nat}.(\alpha, m+k) \to (\beta, m)) \times (\alpha \times \alpha),\\ \qquad n + 2k + 2).\\ \qquad \mathtt{let}\ f = \mathtt{prj}_1\, fp\ \mathtt{in}\\ \qquad \mathtt{let}\ p = \mathtt{prj}_2\, fp\\ \qquad \mathtt{in}\\ \qquad\quad \langle f[n+k+1](\mathtt{prj}_1\, p), f[n](\mathtt{prj}_2\, p)\rangle\end{array}$$

The function first takes four static arguments, two types and two natural numbers: α and β (the domain and codomain types of f), k (the running time of f), and n (the overall finishing time). After taking static arguments, map_pair takes f and p as a pair (curried functions quickly become tedious when writing clock values on each end of every arrow), and specifies the function's starting time to be $n+2k+2$.

The body typechecks as follows: The starting clock, as specified by the abstraction, is $n+2k+2$. The clock only steps for function calls, so the clock is unchanged by extraction of f and p, by the first instantiation of f, and by the projection from p. The first function call expects a starting clock of $(n+k+1)+k$, which it receives once the clock is stepped for the function call. The function call then returns with the clock reading $n+k+1$, as given by f's type. Stepping the clock for the second function call leaves it reading $n+k$, the expected starting clock for the second function call, which then returns with the clock reading n.

Therefore, map_pair is given the type:

$$\begin{array}{l}\forall\alpha{:}\mathsf{Type}, \beta{:}\mathsf{Type}, k{:}\mathsf{Nat}, n{:}\mathsf{Nat}.\\ \quad ((\forall m{:}\mathsf{Nat}.(\alpha, m+k) \to (\beta, m)) \times (\alpha \times \alpha), n+2k+2)\\ \quad \to (\beta \times \beta, n)\end{array}$$

Effects systems At alternative formulation that one could consider would use an effects type system [19] to track the passage of time. Such a system would allow for a simpler notation for function types ($\mathtt{int} \xrightarrow{3} \mathtt{int}$) as opposed to $\forall n.\, (\mathtt{int}, n+3) \to (\mathtt{int}, n)$) and would eliminate the effort seen in the previous example in matching up clock readings.

We did not adopt an effects formulation because, although such a formulation would works quite neatly for time, it does not neatly generalize to resources that can be recovered, such as space. For example, if we interpret resources to mean space, the type $\forall n.\, (\mathtt{int}, n+15) \to (\mathtt{int}, n+12)$ specifies a function that allocates as many as 15 units of space but deallocates all but 3 of those, and so must be called with at least 15 units of space. There is no convenient way to specify such a function using a conventional effects system. When resources cannot be recovered, such a type is pointless (one would always prefer the more flexible $\forall n.\, (\mathtt{int}, n+3) \to (\mathtt{int}, n)$), but when they can, such types draw clearly important distinctions. Nevertheless, in this paper we will adopt the effects notation as a convenient shorthand:

Notation 2.1 *We write* $\tau_1 \xrightarrow{n} \tau_2$ *to mean* $\forall m{:}\mathsf{Nat}.\, (\tau_1, m+n) \to (\tau_2, m)$.

For example, map_pair's type may be abbreviated:

$$\forall\alpha{:}\mathsf{Type}, \beta{:}\mathsf{Type}, k{:}\mathsf{Nat}.\, ((\alpha \xrightarrow{k} \beta) \times (\alpha \times \alpha)) \xrightarrow{2k+2} (\beta \times \beta)$$

2.1 Variable-time functions

The programming idiom seen so far is pleasantly simple, but it supports only constant-time functions, which renders it insufficient for most real programs. Even when whole programs run in constant time, they still very often contain variable-time subroutines. In order to support real programs, we must permit a function's running time to depend on its arguments. Since this involves type expressions referring to values, we require some notion of dependent type.

One possible design is simply to add dependent types to the system directly. It is possible that a workable design could be constructed using such an approach (following Xi and Pfenning [21], for example), but we adopted another approach to better leverage the type structure already developed for TAL [3], and in order to keep the type system as simple as possible. In our approach, we take sum and inductive kinds and polymorphism as our primitive constructs, and use them to construct a form of synthetic dependent type. As a result, our programming language is simpler, but the programming idiom is more complex. This is reasonable in keeping with our intention for this language to be primarily machine manipulated; we do however include dependent types in our compiler's source language, PopCron (Section 4).

Our approach to dependent types is based on a device from Crary and Weirich [3] (hereafter, CW), for the compiler intermediate language LX. CW showed how, using sum and inductive kinds, to construct run-time values that represent static type information. This made it possible to perform intensional type analysis [7] while inspecting only values. Here we use the same device for the opposite purpose, to construct static representations of run-time values. We can then make a function's running time depend on the static representation of an argument, rather than on the argument itself.

We begin with a representative example to illustrate how this works. Suppose we wish to implement a map_tree function. The running time of map_tree depends on the size of the tree: If k is the running time of the argument function and t is the tree argument, then map_tree's running time is $\mathsf{size}(t) \cdot (k+3)$ (one call to the mapping function and two recursive calls for each node). Using an ordinary dependent type, we might write map_tree's type as something like:

$$\forall\alpha{:}\mathsf{Type}, \beta{:}\mathsf{Type}, k{:}\mathsf{Nat}. \\ (\alpha \xrightarrow{k} \beta, t : \mathtt{tree}\,\alpha) \xrightarrow{\mathsf{size}(t)\cdot(k+3)} \mathtt{tree}\,\beta$$

In our idiom, we add an additional static argument (s) that represents the argument t, and we compute the running time based on that static argument:

$$\forall\alpha{:}\mathsf{Type}, \beta{:}\mathsf{Type}, k{:}\mathsf{Nat}, s{:}\mathtt{TreeRep}. \\ ((\alpha \xrightarrow{k} \beta) \times \mathtt{tree}(s)(\alpha)) \xrightarrow{\mathtt{cost}(s)} \mathtt{tree}(s)(\beta)$$

We define TreeRep using an inductive kind and cost using primitive recursion over that inductive kind. In this section we will use an informal notation for each of these; we show how the example is formalized in the next section:

```
kind TreeRep = Leaf
             | Node of TreeRep * TreeRep

fun cost(Leaf) = 0
  | cost(Node (s1, s2)) =
       cost(s1) + cost(s2) + k + 3
```

The example is simplified by the fact that s needs only to represent the shape of the tree, and not its contents, allowing the same representation to be used for the argument and result trees. If the mapping function were a variable-time function, instead of constant-time k, the TreeRep kind would need to be augmented to supply information about the data items lying at each node. Also note that we have simplified the definition of cost by including the variable k free, rather than as an argument.

The most delicate aspect is the definition of $\mathtt{tree}(s)$ as a parameterized recursive type that includes only those trees represented by the argument, s. A simple and naively appealing definition is:

```
type wrong_tree(s)(α) =
   (case s of
       Leaf => unit + void
     | Node (s1, s2) =>
          void + (α * wrong_tree(s1)(α)
                    * wrong_tree(s2)(α)))
```

Unfortunately, this definition makes map_tree unimplementable, as s, the argument to tree, will nearly always be abstract. In such cases, $\mathtt{wrong\_tree}(s)(\alpha)$ will be a useless, irreducible type. Instead, following CW, if we implement it as (again using an informal notation):

```
type tree(s)(α) =
   (case s of
       Leaf => unit
     | Node (s1, s2) => void) +
   (case s of
       Leaf => void
     | Node (s1, s2) =>
          α * tree(s1)(α) * tree(s2)(α))
```

Given a member of t of $\mathtt{tree}(s)(\alpha)$, we can always case-split t. Suppose t is a left injection, $\mathtt{inj}_1\, t'$. Then we know that s is necessarily Leaf, because if it were Node, then t' would have type void, which is impossible. LXres permits us to propagate this new information back into the type system and refine s according to whatever we learn. In this manner, static representations can keep up with their dynamic counterparts.

With these definitions, we can implement map_tree as shown below. The propagation of information into the type system is conducted by a *virtual case* construct (described in Section 3), but for the sake of clarity in this informal account, we allow that propagation to be implicit in the example.

$$
\begin{array}{l}
\mathtt{fix}\ \mathsf{map\_tree} = \\
\Lambda\alpha{:}\mathsf{Type}, \beta{:}\mathsf{Type}, k{:}\mathsf{Nat}, s{:}\mathtt{TreeRep}, n{:}\mathsf{Nat}. \\
\quad \lambda(\mathit{ft} : (\alpha \xrightarrow{k} \beta) \times \mathtt{tree}(s)(\alpha), n + \mathtt{cost}(s)). \\
\quad\quad \mathtt{let}\,\langle f : \alpha \xrightarrow{k} \beta, t : \mathtt{tree}(s)(\alpha)\rangle = \mathit{ft}\,\mathtt{in} \\
\quad\quad \mathtt{case}\ t\ \mathtt{of} \\
\quad\quad\quad \mathtt{inj}_1\, t' \Rightarrow \mathtt{inj}_1 * \\
\quad\quad\quad \mathtt{inj}_2\, t' \Rightarrow \\
\quad\quad\quad\quad \mathtt{let}\,\mathtt{Node}(s_1, s_2) = s\,\mathtt{in} \\
\quad\quad\quad\quad \mathtt{let}\,\langle x : \alpha, t_1 : \mathtt{tree}(s_1)(\alpha), t_2 : \mathtt{tree}(s_2)(\alpha)\rangle = t'\,\mathtt{in} \\
\quad\quad\quad\quad \mathtt{let}\,x' = f[n + \mathtt{cost}(s_1) + \mathtt{cost}(s_2) + 2](x)\,\mathtt{in} \\
\quad\quad\quad\quad \mathtt{let}\,t_1' = \mathsf{map\_tree}[\alpha, \beta, k, s_1, \\
\quad\quad\quad\quad\quad\quad\quad\quad n + \mathtt{cost}(s_2) + 1]\langle f, t_1\rangle\,\mathtt{in} \\
\quad\quad\quad\quad \mathtt{let}\,t_2' = \mathsf{map\_tree}[\alpha, \beta, k, s_2, n]\langle f, t_2\rangle \\
\quad\quad\quad\quad \mathtt{in} \\
\quad\quad\quad\quad\quad \mathtt{inj}_2\langle x', t_1', t_2'\rangle
\end{array}
$$

The function typechecks as follows: We begin with a function f that runs in k steps, an abstract tree representation s, a tree t matching that representation, and $n + \mathtt{cost}(s)$ on the clock. The tree t is a member of a sum type, so we can case analyze it. In the first case, t is $\mathtt{inj}_1\, t'$ and therefore, as argued above, s is Leaf. Consequently, $\mathtt{cost}(s) = 0$ and the clock reads n. The result value $(\mathtt{inj}_1 *)$ has type $\mathtt{tree}(\mathtt{Leaf})(\alpha)$ and constructing it takes no steps off the clock, so the function's postcondition is satisfied.

In the second case, t is $\mathtt{inj}_2\, t'$ and therefore, since t' cannot have type void, we may infer that s is $\mathtt{Node}(s_1, s_2)$ (for some tree representations s_1 and s_2) and that t' has type $\alpha \times \mathtt{tree}(s_1)(\alpha) \times \mathtt{tree}(s_2)(\alpha)$. A let expression binds x, t_1 and t_2 to the respective components of the node. We then compute the components x', t_1' and t_2' of a new node in three steps:

1. With the clock reading $n + \mathtt{cost}(s) = n + \mathtt{cost}(s_1) + \mathtt{cost}(s_2) + k + 3$, we instantiate f with the finishing time $n + \mathtt{cost}(s_1) + \mathtt{cost}(s_2) + 2$. Thus f expects a starting time of $n + \mathtt{cost}(s_1) + \mathtt{cost}(s_2) + 2 + k$, which is satisfied once the clock is stepped for the function call.

2. With the clock now reading $n + \mathtt{cost}(s_1) + \mathtt{cost}(s_2) + 2$, we instantiate map_tree with a tree representation s_1 and a finishing time $n + \mathtt{cost}(s_2) + 1$. Thus map_tree expects a starting time of $n + \mathtt{cost}(s_2) + 1 + \mathtt{cost}(s_1)$, which is satisfied once the clock is stepped. The argument, t_1 has type $\mathtt{tree}(s_1)(\alpha)$, so the function call is well-typed.

3. With the clock now reading $n + \mathtt{cost}(s_2) + 1$, we instantiate map_tree with s_2 and a finishing time of n. The

starting time, $n + \mathtt{cost}(s_2)$, is satisfied and the argument has the appropriate type, so the function call is again well-typed.

The function concludes by assembling these components into a tree node with type $\mathtt{tree}(\mathtt{Node}(s_1, s_2))(\alpha)$ and the clock reads n, so the function's postcondition is satisfied.

2.2 Discussion

The map_tree example is typical of the use of our system. First, one identifies one or more *metric arguments* on which a function's running time depends. In this example, the running time depends on both the running time of f and the size of t, so both f and t are metric arguments.

Second, one defines representations for the metric arguments that are detailed enough both to compute the cost function and to define *enforcement types* (such as $\mathtt{tree}(s)$) that contain only values that match the representative. The representation for the f argument was simple; all that needed to be known was its running time k and an enforcement type $\alpha \xrightarrow{k} \beta$ was easily defined. The t argument is more complicated (and often more typical). We would be happy to represent it by only its size, but our type system is incapable of constructing an enforcement type for a tree given only its size (more on this below), and so consequently we needed to specify the tree's entire shape.

Finally, after defining a cost function and an enforcement type, the function itself is written. Two main issues arise in this process:

- The cost must be woven through the function; this is done by correct choices of finishing times for function calls and by wasting extra clock cycles in order to make parallel branches match. (The latter was not necessary in the map_tree example.)

- Representations must be supplied for all metric arguments to inner function calls. Constructing such representations requires access to representations of all components from which those arguments are built. When a component is part of one of the function's own metric arguments, the desired representation is easily obtained from that argument's representation. (In this map_tree example, the needed representations were obtained by decomposing s into $\mathtt{Node}(s_1, s_2)$ once it was determined that t was a node.)

  When a component is the result of another function call, that function's type must specify the representation of that return value. For example, map_tree specified that its result had the same specification as its argument. More generally, the result could have a different specification given by a static function. For example, if map_tree also possibly restructured the tree, it would need to have the type

$$\begin{array}{l}\forall \alpha{:}\mathsf{Type}, \beta{:}\mathsf{Type}, k{:}\mathsf{Nat}, s{:}\mathtt{TreeRep}.\\ \quad ((\alpha \xrightarrow{k} \beta) \times \mathtt{tree}(s)(\alpha))\\ \qquad \xrightarrow{\mathtt{cost}(s)} \mathtt{tree}(\mathtt{newrep}(s))(\beta)\end{array}$$

  where $\mathtt{newrep}$ is a static function with kind $\mathtt{TreeRep} \to \mathtt{TreeRep}$. Such functions with specified results are more limited in what they can do than ordinary functions, because static functions are required to be primitive recursive and therefore the specified functions must be primitive recursive, at least as far as the structure described by the representations are concerned. This requirement, which we discuss further in the next section, is necessary to ensure decidability of type checking.

  In some cases, an argument may be a metric argument to an inner function call even when it does not affect the total running time of the outer function. For example, suppose a function broke a fixed-length array into two variable-sized pieces based on an integer argument, and then mapped a constant-time function over each piece. The total running time does not depend on the integer argument, but each internal call's time does. In such cases, the "phantom" dependency must be treated as a metric argument and a representation must be required.

  A similar issue arises when an inner function's result is a metric argument to another inner function, but that result does not affect the total remaining running time of the outer function. In this case, a representation must again be obtained, but since there is no need to link this representation back to the total cost, it can be existentially quantified rather than specified by a static function. This case is more typical than the case of functions that require specified results.

Returning to the topic of enforcement types, the aspect of trees that makes it impossible to define an enforcement type for them given only a size is that trees have multiple spines. That is, given a total size, the enforcement type does not know how much size to place in the left subtree and how much to place in the right. Therefore, the representation must also specify how much to place in each subtree, and by doing so inductively it specifies the tree's entire shape. Multiple-spined data structures are quite common, so it is common to require representations of entire data structures. Consequently, our prototype compiler makes no effort to optimize representations, instead providing (nearly) full representations for all metric arguments (other than functions). However, it is important to emphasize that the static representations are *not* constructed at run time, so the cost of over-representing a metric argument is paid at verification time only.

An alternative strategy that could permit much more succinct representations at the cost of a more complicated type system would be to use union types to enumerate the possible divisions of the total size. We have not explored the ramifications of such a design, but it seems likely that it would complicate typechecking.

3 A Language for Resource Bound Certification

In this section we discuss LXres and its semantics. We present the constructor and term levels individually, concentrating discussion on the novel features of each. The syntax of LXres (shown in Figures 1 and 2) is based on Girard's F_ω [6, 5] augmented mainly by a rich programming language at the constructor level, constructor refinement operators at the term level, and resource bounds in types and terms. LXres is very similar to CW's LX, the difference being that LXres includes virtual clocks and a primitive natural number kind in support of those clocks. The full static and operational semantics of LXres are given in Appendices A and B.

(kinds)	k	$::= \mathsf{Type} \mid \mathsf{Nat} \mid 1 \mid k_1 \to k_2 \mid k_1 \times k_2 \mid k_1 + k_2 \mid j \mid \mu j.k$	
(constructors)	c, τ	$::= * \mid \alpha \mid \lambda\alpha{:}k.c \mid c_1 c_2$	unit, variables and functions
		$\mid \langle c_1, c_2\rangle \mid \mathtt{prj}_1\, c \mid \mathtt{prj}_2\, c$	products
		$\mid \mathtt{inj}_1^{k_1+k_2}\, c \mid \mathtt{inj}_2^{k_1+k_2}\, c \mid \mathtt{case}(c, \alpha_1.c_1, \alpha_2.c_2)$	sums
		$\mid \mathtt{fold}_{\mu j.k}\, c \mid \mathtt{pr}(j, \alpha{:}k, \beta{:}j \to k'.c)$	primitive recursion
		$\mid \overline{n} \mid c_1 \dotplus c_2 \mid \mathtt{prnat}(\alpha, \beta{:}k.c_1; c_2)$	natural numbers
		$\mid (\tau_1, c_1) \to (\tau_2, c_2) \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2$	types
		$\mid \forall\alpha{:}k.\tau \mid \exists\alpha{:}k.\tau \mid \mathtt{unit} \mid \mathtt{void} \mid \mathtt{rec}_k(c_1, c_2)$	types

Figure 1: LXres kinds and constructors

3.1 Kinds and constructors

The constructor and kind levels, shown in Figure 1, contain both base constructors of kind Type (called types) for classifying terms, and a variety of programming constructs for computing types. In addition to the variables and lambda abstractions of F_ω, LXres also includes a unit kind, products, sums, and the usual introduction and elimination constructs for those kinds. LXres also includes a natural number kind discussed further in Section 3.3.

We denote the simultaneous, capture-avoiding substitution of $E_1, \ldots, E_n$ for $X_1, \ldots, X_n$ in E by $E[E_1, \ldots, E_n / X_1, \ldots, X_n]$. As usual, we consider alpha-equivalent expressions to be identical. A few constructs ($\mathtt{inj}_i$, $\mathtt{fold}$, $\mathtt{pr}$, and $\mathtt{rec}$) are labeled with kinds to assist in kind checking; we will omit such kinds when they are clear from context. When a constructor is intended to have kind Type, we often use the metavariable τ.

To support computing with static representations, LXres includes kind variables (j) and inductive kinds ($\mu j.k$). A prospective inductive kind $\mu j.k$ will be well-formed provided that j appears only positively within k. Inductive kinds are formed using the introductory operator $\mathtt{fold}_{\mu j.k}$, which coerces constructors from kind $k[\mu j.k/j]$ to kind $\mu j.k$. For example, consider the kind Tlist of lists of types, defined as $\mu j.(1{+}\mathsf{Type}{\times}j)$. The constructor $(\mathtt{inj}_1^{1+\mathsf{Type}\times\mathsf{Tlist}} *)$ has kind $(1{+}\mathsf{Type}{\times}j)[\mathsf{Tlist}/j]$. Therefore $\mathtt{fold}_{\mathsf{Tlist}}(\mathtt{inj}_1^{1+\mathsf{Type}\times\mathsf{Tlist}} *)$ has kind Tlist.

Inductive kinds are eliminated using the primitive recursion operator $\mathtt{pr}$. Intuitively, $\mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c)$ may be thought of as a recursive function with domain $\mu j.k$ in which α stands for the argument unfolded and φ recursively stands for the full function. However, in order to ensure that constructor expressions always terminate, we restrict $\mathtt{pr}$ to define only primitive recursive functions. Informally speaking, a function is primitive recursive if it can only call itself recursively on a subcomponent of its argument. Following Mendler [12], we ensure this using abstract kind variables. Since α stands for the argument unfolded, we could consider it to have the kind $k[\mu j.k/j]$, but instead of substituting for j in k, we hold j abstract. Then the recursive variable φ is given kind $j \to k'$ (instead of $j[\mu j.k/j] \to k'$) thereby ensuring that φ is called only on a subcomponent of α.

The kind k' in $\mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c)$ is permitted to contain (positive) free occurrences of j. In that case, the function's result kind employs the substitution for j that was internally eschewed. Hence, the result kind of the above constructor is $k'[\mu j.k/j]$. This is useful so that some part of the argument may be passed through without φ operating on it. As a particularly useful application, we can define the constructor $\mathtt{unfold}_{\mu j.k}$ with kind $\mu j.k \to k[\mu j.k/j]$ to be $\mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k.\alpha)$.

For example, given a constructor with kind Tlist, we can use primitive recursion to construct the tuple type built from the types in the list (using an informal, expanded notation for $\mathtt{case}$):

$$\begin{aligned}\mathsf{tuple} \stackrel{\text{def}}{=}\ & \mathtt{pr}(j, \alpha{:}1{+}\mathsf{Type}{\times}j, \varphi{:}j \to \mathsf{Type}.\\ & \quad \mathtt{case}\ \alpha\ \mathtt{of}\\ & \quad\quad \mathtt{inj}_1\, \beta \Rightarrow \mathtt{unit}\\ & \quad\quad \mathtt{inj}_2\, \gamma \Rightarrow \mathtt{prj}_1\, \gamma \times \varphi(\mathtt{prj}_2\, \gamma))\end{aligned}$$

Suppose we apply tuple to the list $\mathtt{[int]}$ (that is, the encoding $\mathtt{fold}(\mathtt{inj}_2\langle\mathtt{int}, \mathtt{fold}(\mathtt{inj}_1 *)\rangle)$). By unrolling the $\mathtt{pr}$ expression, we may show:

$$\begin{aligned}& (\mathtt{pr}(j, \alpha{:}1{+}\mathsf{Type}{\times}j, \varphi{:}j \to \mathsf{Type}.\\ & \quad \mathtt{case}\ \alpha\ \mathtt{of}\\ & \quad\quad \mathtt{inj}_1\, \beta \Rightarrow \mathtt{unit}\\ & \quad\quad \mathtt{inj}_2\, \gamma \Rightarrow \mathtt{prj}_1\, \gamma \times \varphi(\mathtt{prj}_2\, \gamma)))\ \mathtt{[int]}\\[1ex] & = \mathtt{case}\ (\mathtt{inj}_2\langle\mathtt{int}, \mathtt{fold}(\mathtt{inj}_1 *)\rangle)\ \mathtt{of}\\ & \quad\quad \mathtt{inj}_1\, \beta \Rightarrow \mathtt{unit}\\ & \quad\quad \mathtt{inj}_2\, \gamma \Rightarrow \mathtt{prj}_1\, \gamma \times \mathsf{tuple}(\mathtt{prj}_2\, \gamma)\\[1ex] & = \mathtt{int} \times (\mathsf{tuple}(\mathtt{fold}(\mathtt{inj}_1 *)))\\ & \quad (\text{that is, } \mathtt{int} \times \mathsf{tuple}(\mathtt{[]}))\\[1ex] & = \mathtt{int} \times (\mathtt{case}\ (\mathtt{inj}_1 *)\ \mathtt{of}\\ & \quad\quad\quad \mathtt{inj}_1\, \beta \Rightarrow \mathtt{unit}\\ & \quad\quad\quad \mathtt{inj}_2\, \gamma \Rightarrow \mathtt{prj}_1\, \gamma \times \mathsf{tuple}(\mathtt{prj}_2\, \gamma))\\[1ex] & = \mathtt{int} \times \mathtt{unit}\end{aligned}$$

The unrolling process is formalized by the following constructor equivalence rule (the relevant judgment forms are summarized in Figure 5):

$$\frac{\begin{array}{cc}\Delta \vdash c' : k[\mu j.k/j] & \Delta, j \vdash k'\ \mathsf{kind}\\ \Delta, j, \alpha{:}k, \varphi{:}j \to k' \vdash c : k' & \Delta \vdash \mu j.k\ \mathsf{kind}\end{array}}{\begin{array}{l}\Delta \vdash \mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c)(\mathtt{fold}_{\mu j.k}\, c') =\\ \quad c[\mu j.k, c', \mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c)/j, \alpha, \varphi]\\ \quad\quad : k'[\mu j.k/j]\end{array}}\ (j \text{ only positive in } k' \text{ and } j, \alpha, \varphi \notin \Delta)$$

Notation 3.1 If k_1 is of the form $\mu j.k$, then we write $k_1[k_2]$ to mean $k[k_2/j]$. For example, Tlist[Tlist] abbreviates $1 + \mathsf{Type} \times \mathsf{Tlist}$.

3.2 Types and terms

The syntax of LXres terms is given in Figure 2. Most LXres terms are standard, including the usual introduction and

(terms)	e	$::= * \mid x \mid \lambda(x{:}\tau, c).e \mid e_1 e_2$	unit, variables, functions
		$\mid \texttt{waste}[c]e$	clock advance
		$\mid \langle e_1, e_2\rangle \mid \texttt{prj}_1\, e \mid \texttt{prj}_2\, e$	products
		$\mid \texttt{inj}_1^{\tau_1+\tau_2}\, e \mid \texttt{inj}_2^{\tau_1+\tau_2}\, e \mid \texttt{case}(e, x_1.e_1, x_2.e_2)$	sums
		$\mid \Lambda\alpha{:}k.v \mid e[c] \mid \texttt{fix}\, f{:}\tau.v$	constructor abstractions and recursion
		$\mid \texttt{pack}\, e\, \texttt{as}\, \exists\alpha{:}k.\tau\ \texttt{hiding}\, c \mid \texttt{unpack}\, \langle\alpha, x\rangle = e_1\, \texttt{in}\, e_2$	existential packages
		$\mid \texttt{fold}_{\texttt{rec}_k(c,c')}\, e \mid \texttt{unfold}\ e$	parameterized recursive types
		$\mid \texttt{vcase}_{\tau\triangleright c}(c, \beta.\, e, \gamma.\, \texttt{dead}\, v) \mid \texttt{vcase}_{\tau\triangleright c}(c, \beta.\, \texttt{dead}\, v, \gamma.\, e)$	constructor refinement operations
		$\mid \texttt{let}_{\tau\triangleright c}\, \langle\beta, \gamma\rangle = c\ \texttt{in}\, e \mid \texttt{let}_{\tau\triangleright c}\, (\texttt{fold}\, \beta) = c\ \texttt{in}\, e$	constructor refinement operations

Figure 2: LXres terms

$$\frac{\Delta \vdash \tau_1 : \mathsf{Type} \quad \Delta \vdash c_1 : \mathsf{Nat} \quad \Delta \vdash \tau_2 : \mathsf{Type} \quad \Delta \vdash c_2 : \mathsf{Nat}}{\Delta \vdash (\tau_1, c_1) \to (\tau_2, c_2) : \mathsf{Type}}$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{Type} \quad \Delta \vdash c_1 : \mathsf{Nat} \quad \Delta; (\Gamma, x{:}\tau_1); c_1 \vdash e : \tau_2 \triangleright c_2}{\Delta; \Gamma; c \vdash \lambda(x{:}\tau_1, c_1).e : (\tau_1, c_1) \to (\tau_2, c_2) \triangleright c} \quad (x \notin \Gamma)$$

$$\frac{\begin{array}{c}\Delta; \Gamma; c \vdash e_1 : (\tau_1, c_1) \to (\tau_2, c_2) \triangleright c' \\ \Delta; \Gamma; c' \vdash e_2 : \tau_1 \triangleright (c_1 \dot{+} 1)\end{array}}{\Delta; \Gamma; c \vdash e_1 e_2 : \tau_2 \triangleright c_2}$$

$$\frac{\Delta; \Gamma; c \vdash e : \tau \triangleright (c_1 \dot{+} c_2)}{\Delta; \Gamma; c \vdash \texttt{waste}[c_1]e : \tau \triangleright c_2}$$

Figure 3: Virtual clock rules

elimination forms for products, sums, unit, and universal and existential types.

Function types and lambda abstractions are written with virtual clock specifications, as discussed in Section 2. A few representative rules governing virtual clocks appear in Figure 3, in which the typing judgment $\Delta; \Gamma; c \vdash e : \tau \triangleright c'$ states that (in constructor context Δ and value context Γ) the term e has type τ, and given a clock reading c it finishes evaluating with the clock reading c'.

Constructor abstractions are limited by a value restriction to avoid unsoundness in the presence of effects. The value forms are given in Appendix B. Recursive functions are expressible using `fix` terms, the bodies of which are syntactically restricted to be functions (usually polymorphic) by their typing rule (Appendix A). As at the constructor level, some constructs are labeled with types or clocks to assist in type checking; we omit these when clear from context.

Parameterized recursive types are written $\texttt{rec}_k(c_1, c_2)$, where k is the parameter kind and c_1 is a type constructor with kind $(k \to \mathsf{Type}) \to (k \to \mathsf{Type})$. Intuitively, c_1 recursively defines a type constructor with kind $k \to \mathsf{Type}$, which is then instantiated with the parameter c_2 (having kind k). Thus, members of $\texttt{rec}_k(c_1, c_2)$ unfold into the type $c_1(\lambda\alpha{:}k.\, \texttt{rec}_k(c_1, \alpha))c_2$, and fold the opposite way. The special case of non-parameterized recursive types are defined as $\texttt{rec}(\alpha.\tau) = \texttt{rec}_1(\lambda\varphi{:}1 \to \mathsf{Type}.\, \lambda\beta{:}1.\, \tau[\varphi(*)/\alpha], *)$. Unlike inductive kinds, no positivity condition is imposed on recursive types.

Refinement The most novel features of the LXres term language are the constructor refinement operations. The main refinement operations are the two *virtual case* constructs, written $\texttt{vcase}(c, \beta.\, e, \gamma.\, \texttt{dead}\, v)$ and dually. These operations may been thought of as branching operations for sum kinds: if c normalizes to $\texttt{inj}_1(c')$, then the term $\texttt{vcase}(c, \beta.\, e, \gamma.\, \texttt{dead}\, v)$ evaluates to $e[c'/\beta]$.

However, constructors are *static* components of the program only, and therefore may not be relied upon at run time. Consequently, the semantics of $\texttt{vcase}(c, \beta.\, e, \gamma.\, \texttt{dead}\, v)$ dictates that it always reduces to the first branch, and the second branch is dead code. To avoid unsoundness, the typing rule for `vcase` requires that its dead branch be a value with type `void`, and thus that branch is statically known to be dead code. Somewhat surprisingly, it may be shown [3] that this virtual case construct provides the same expressive power as an unrestricted case construct would.

In the left branch of a virtual case, the constructor being branched on is determined to be a left injection, and conversely in the right branch. When that constructor is a variable, the `vcase` typing rule propagates this information back into the type system by substituting for that variable:

$$\frac{\begin{array}{l}\Delta, \beta{:}k_1, \Delta'; \Gamma[\texttt{inj}_1\, \beta/\alpha]; c_1[\texttt{inj}_1\, \beta/\alpha] \vdash \\ \qquad e[\texttt{inj}_1\, \beta/\alpha] : \tau[\texttt{inj}_1\, \beta/\alpha] \triangleright c_2[\texttt{inj}_1\, \beta/\alpha] \\ \Delta, \gamma{:}k_2, \Delta'; \Gamma[\texttt{inj}_2\, \gamma/\alpha]; c_1[\texttt{inj}_2\, \gamma/\alpha] \vdash \\ \qquad v[\texttt{inj}_2\, \gamma/\alpha] : \texttt{void} \triangleright c_2' \\ \Delta, \alpha{:}k_1{+}k_2, \Delta' \vdash c = \alpha : k_1 + k_2\end{array}}{\Delta, \alpha{:}k_1{+}k_2, \Delta'; \Gamma; c_1 \vdash \texttt{vcase}_{\tau\triangleright c_2}(c, \beta.\, e, \gamma.\, \texttt{dead}\, v) : \tau \triangleright c_2} \quad (\beta, \gamma \notin \Delta)$$

Within the branches, types that depend on α can be reduced using the new information. For example, if x has type $\texttt{case}(\alpha, \beta.\texttt{int}, \beta.\texttt{void})$, its type can be reduced in each branch, allowing its use as an integer in the active branch and as `void` value in the dead branch.

In order for LXres to enjoy the subject reduction property, we also require a *trivialization* rule [4] for `vcase`, for use when the argument is a sum introduction:

$$\frac{\Delta \vdash c = \texttt{inj}_1\, c' : k_1 + k_2 \quad \Delta; \Gamma; c_1 \vdash e[c'/\beta] : \tau \triangleright c_2}{\Delta; \Gamma; c_1 \vdash \texttt{vcase}_{\tau\triangleright c_2}(c, \beta.\, e, \gamma.\, \texttt{dead}\, v) : \tau \triangleright c_2}$$

Path refinement There may also be useful refinement to perform when the constructor to be branched on is not a variable. For example, suppose α has kind $(1{+}1) \times \mathsf{Type}$ and

$$
\begin{array}{l}
\mathtt{TreeRep} = \mu j.(1 + j \times j) \\
\mathtt{cost} \quad = \mathtt{pr}(j, \alpha{:}\mathtt{TreeRep}[j], \varphi{:}j \to \mathsf{Nat}. \\
\qquad\qquad\qquad \mathtt{case}\,\alpha\,\mathtt{of} \\
\qquad\qquad\qquad\quad \mathtt{inj}_1\,\beta \Rightarrow \overline{0} \\
\qquad\qquad\qquad\quad \mathtt{inj}_2\,\beta \Rightarrow \\
\qquad\qquad\qquad\qquad \varphi(\mathtt{prj}_1\,\beta) \dotplus \varphi(\mathtt{prj}_2\,\beta) \dotplus k \dotplus \overline{3}) \\
\mathtt{tree}(s)(\alpha) = \\
\mathtt{rec}_{\mathtt{TreeRep}}(\lambda\varphi{:}\mathtt{TreeRep} \to \mathsf{Type}.\, \lambda\beta{:}\mathtt{TreeRep}. \\
\qquad\qquad (\mathtt{case}\,\mathtt{unfold}\,\beta\,\mathtt{of} \\
\qquad\qquad\quad \mathtt{inj}_1\,\gamma \Rightarrow * \\
\qquad\qquad\quad \mathtt{inj}_2\,\gamma \Rightarrow \mathtt{void}) + \\
\qquad\qquad (\mathtt{case}\,\mathtt{unfold}\,\beta\,\mathtt{of} \\
\qquad\qquad\quad \mathtt{inj}_1\,\gamma \Rightarrow \mathtt{void} \\
\qquad\qquad\quad \mathtt{inj}_2\,\gamma \Rightarrow \\
\qquad\qquad\qquad \alpha \times \varphi(\mathtt{prj}_1\,\gamma)(\alpha) \times \varphi(\mathtt{prj}_2\,\gamma)(\alpha)), \\
\qquad s)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{map\_tree} = \\
\mathtt{fix}\,\mathsf{map\_tree}: \\
\quad (\forall\alpha{:}\mathsf{Type}, \beta{:}\mathsf{Type}, k{:}\mathsf{Nat}, s{:}\mathtt{TreeRep}. \\
\qquad\qquad ((\alpha \xrightarrow{k} \beta) \times \mathtt{tree}(s)(\alpha)) \xrightarrow{\mathtt{cost}(s)} \mathtt{tree}(s)(\beta)). \\
\Lambda\alpha{:}\mathsf{Type}, \beta{:}\mathsf{Type}, k{:}\mathsf{Nat}, s{:}\mathtt{TreeRep}, n{:}\mathsf{Nat}. \\
\quad \lambda(ft : (\alpha \xrightarrow{k} \beta) \times \mathtt{tree}(s)(\alpha), n \dotplus \mathtt{cost}(s)). \\
\qquad \mathtt{let}\,\langle f : \alpha \xrightarrow{k} \beta, t : \mathtt{tree}(s)(\alpha)\rangle = ft\,\mathtt{in} \\
\qquad \mathtt{case}\,\mathtt{unfold}\,t\,\mathtt{of} \\
\qquad\quad \mathtt{inj}_1\,t' \Rightarrow \\
\qquad\qquad \mathtt{let}\,\mathtt{fold}(s') = s\,\mathtt{in} \\
\qquad\qquad \mathtt{vcase}\,s'\,\mathtt{of} \\
\qquad\qquad\quad \mathtt{inj}_1\,s'' \Rightarrow \mathtt{inj}_1\,* \\
\qquad\qquad\quad \mathtt{inj}_2\,s'' \Rightarrow \mathtt{dead}\,t' \\
\qquad\quad \mathtt{inj}_2\,t' \Rightarrow \\
\qquad\qquad \mathtt{let}\,\mathtt{fold}(s') = s\,\mathtt{in} \\
\qquad\qquad \mathtt{vcase}\,s'\,\mathtt{of} \\
\qquad\qquad\quad \mathtt{inj}_1\,s'' \Rightarrow \mathtt{dead}\,t' \\
\qquad\qquad\quad \mathtt{inj}_2\,s'' \Rightarrow \\
\qquad\qquad\qquad \mathtt{let}\,\langle s_1, s_2\rangle = s''\,\mathtt{in} \\
\qquad\qquad\qquad \mathtt{let}\,\langle x : \alpha, t_1 : \mathtt{tree}(s_1)(\alpha), t_2 : \mathtt{tree}(s_2)(\alpha)\rangle = t'\,\mathtt{in} \\
\qquad\qquad\qquad \mathtt{let}\,x' = f[n \dotplus \mathtt{cost}(s_1) \dotplus \mathtt{cost}(s_2) \dotplus \overline{2}](x)\,\mathtt{in} \\
\qquad\qquad\qquad \mathtt{let}\,t_1' = \mathsf{map\_tree}[\alpha, \beta, k, s_1, \\
\qquad\qquad\qquad\qquad\qquad\qquad n \dotplus \mathtt{cost}(s_2) \dotplus \overline{1}]\langle f, t_1\rangle\,\mathtt{in} \\
\qquad\qquad\qquad \mathtt{let}\,t_2' = \mathsf{map\_tree}[\alpha, \beta, k, s_2, n]\langle f, t_2\rangle \\
\qquad\qquad\qquad \mathtt{in} \\
\qquad\qquad\qquad\quad \mathtt{inj}_2\langle x', t_1', t_2'\rangle
\end{array}
$$

Figure 4: Formalized map_tree example

x has type $\mathtt{case}(\mathtt{prj}_1\,\alpha, \beta.\mathtt{int}, \beta.\mathtt{void})$. When branching on $\mathtt{prj}_1\,\alpha$, we should again be able to consider x an integer or a void-value, but the ordinary `vcase` rule above no longer applies since $\mathtt{prj}_1\,\alpha$ is not a variable. This is solved using the product refinement operation, $\mathtt{let}_{\tau \triangleright c}\,\langle\beta, \gamma\rangle = \alpha\,\mathtt{in}\,e$. Like `vcase`, the product refinement operation substitutes everywhere for α:

$$
\frac{\begin{array}{l}\Delta, \beta{:}k_1, \gamma{:}k_2, \Delta'; \Gamma[\langle\beta,\gamma\rangle/\alpha] \triangleright c_1[\langle\beta,\gamma\rangle/\alpha] \vdash \\ \qquad e[\langle\beta,\gamma\rangle/\alpha] : \tau[\langle\beta,\gamma\rangle/\alpha] \triangleright c_2[\langle\beta,\gamma\rangle/\alpha] \\ \Delta, \alpha{:}k_1 \times k_2, \Delta' \vdash c = \alpha : k_1 \times k_2\end{array}}{\Delta, \alpha{:}k_1 \times k_2, \Delta'; \Gamma; c_1 \vdash \mathtt{let}_{\tau \triangleright c_2}\,\langle\beta, \gamma\rangle = c\,\mathtt{in}\,e : \tau \triangleright c_2} \quad (\beta, \gamma \notin \Delta)
$$

A similar refinement operation exists for inductive types, and each operation also has a trivialization rule similar to those of `vcase`.

We may use these refinement operations to turn paths into variables and thereby take advantage of `vcase`. For example, suppose α has kind $\mathsf{Tlist} \times \mathsf{Tlist}$ and we wish to branch on $\mathtt{unfold}\,(\mathtt{prj}_1\,\alpha)$. We do it using product and inductive kind refinement in turn:

$$
\begin{array}{l}
\mathtt{let}\,\langle\beta_1, \beta_2\rangle = \alpha\,\mathtt{in} \\
\mathtt{let}\,(\mathtt{fold}\,\gamma) = \beta_1\,\mathtt{in} \\
\qquad \mathtt{vcase}(\gamma, \delta.\,e, \delta.\,\mathtt{dead}\,v)
\end{array}
$$

A formalized version of the map_tree example using path refinement appears in Figure 4. For clarity, the example uses a `let` notation on terms as the informal version in Section 2 did.

Non-path refinement Since there is no refinement operation for functions, sometimes a constructor cannot be reduced to a path. Nevertheless, it is still possible to gain some of the benefits of refinement, using a device due to Harper and Morrisett [7]. Suppose φ has kind $\mathsf{Nat} \to (1{+}1)$, x has type $\mathtt{case}(\varphi(4), \beta.\mathtt{int}, \beta.\mathtt{void})$, and we wish to branch on $\varphi(4)$ to learn the type of x. First we use a constructor abstraction to assign a variable α to $\varphi(4)$, thereby enabling `vcase`, and then we use a lambda abstraction to rebind x with type $\mathtt{case}(\alpha, \beta.\mathtt{int}, \beta.\mathtt{void})$:

$$
\begin{array}{l}
(\Lambda\alpha{:}1{+}1.\, \lambda x{:}\,\mathtt{case}(\alpha, \beta.\mathtt{int}, \beta.\mathtt{void}). \\
\quad \mathtt{vcase}(\alpha, \beta.e, \beta.\,\mathtt{dead}\,v))\ [\varphi(4)]\ x
\end{array}
$$

Within e, x will be an integer, and similarly within v. This device has all the expressive power of refinement, but is less efficient because of the need for extra beta-expansions. However, this is the best that can be done with unknown functions.

3.3 Natural numbers

LXres includes a primitive natural number kind, even though natural numbers could be defined as the inductive kind $\mu j.(1+j)$ and addition could be defined using primitive recursion. This is because the inductive kind definition does not provide commutativity and associativity axioms, which are frequently necessary to prove adherence to resource bounds. The problem derives from the absence of an eta-equivalence rule on inductive kinds. For example, the

Judgment	Meaning
$\Delta \vdash k$ kind	k is a well-formed kind
$\Delta \vdash c : k$	c is a valid constructor of kind k
$\Delta \vdash c_1 = c_2 : k$	c_1 and c_2 are equal constructors
$\Delta; \Gamma; c_1 \vdash e : \tau \triangleright c_2$	e is a term of type τ, and, when starting with clock c_1, evaluation of e finishes with clock c_2

Contexts

$$\begin{array}{lcl} \Delta & ::= & \epsilon \mid \Delta, j \mid \Delta, \alpha{:}k \\ \Gamma & ::= & \epsilon \mid \Gamma, x{:}\tau \end{array}$$

Figure 5: Judgments

analog of $\alpha + 0 = 0 + \alpha$ does not hold,

$$\begin{array}{l} \alpha \neq \mathtt{pr}(j, \beta{:}1{+}j, \varphi{:}j \to \mu j.(1{+}j). \\ \quad \mathtt{case}\,\beta\,\mathtt{of} \\ \quad\quad \mathtt{inj}_1\,\gamma \Rightarrow \mathtt{fold}(\mathtt{inj}_1\,\gamma) \\ \quad\quad \mathtt{inj}_2\,\gamma \Rightarrow \mathtt{fold}(\mathtt{inj}_2\,\varphi(\gamma))))\,\alpha \end{array}$$

because the right-hand side is irreducible, even though the equivalence does hold for all closed instances.

By making natural numbers primitive, we can easily add commutativity and associativity axioms. Addition of natural numbers is written $c_1 \dot{+} c_2$ to distinguish it from a sum type. Numerals in LXres are written $\overline{n}$ to distinguish them from numbers in the metatheory, but we will often omit the overbar when there is no possibility of confusion.

Natural numbers are eliminated by primitive recursion. The constructor $\mathtt{prnat}_k(\alpha, \beta.c_1; c_2)$ is a function with kind $\mathsf{Nat} \to k$. On zero it returns c_2, and on $n \dot{+} 1$ it returns the body c_1, in which α stands for n and β stands for the recursively computed value:

$$\frac{\Delta, \alpha{:}\mathsf{Nat}, \beta{:}k \vdash c_1 : k \quad \Delta \vdash c_2 : k \quad \Delta \vdash c : \mathsf{Nat}}{\begin{array}{l}\Delta \vdash \mathtt{prnat}_k(\alpha, \beta.c_1; c_2)(c \dot{+} 1) = \\ \quad c_1[c, \mathtt{prnat}_k(\alpha, \beta.c_1; c_2)(c)/\alpha, \beta] : k\end{array}} \; (\alpha, \beta \notin \Delta)$$

The implemented version of this type system, TALres, also provides a variety of other primitive operations (*e.g.*, minimum, proper subtraction, and multiplication) and appropriate axioms for those operations.

3.4 Properties of LXres

The judgments of the static semantics of LXres appear in Figure 5. The important properties to show are decidable type checking and type safety. Due to space considerations, we do not present proofs of these properties here; the proofs are nearly identical to those for LX [3]. For typechecking, the challenging part is deciding equality of type constructors. We do this using a normalize, sort, and compare algorithm employing a reduction relation extracted from the equality rules. The reduction relation is constructed in the obvious manner except for commutativity and associativity, where it simply moves natural number literals outward. A second phase of the algorithm sorts addition expressions to account for full commutativity and associativity.

Lemma 3.2 *Reduction of well-formed constructors is strongly normalizing, confluent, preserves kinds, and is respected by equality.*

Strong normalization is proven using Mendler's variation on Girard's method [12]. Given Lemma 3.2, it is easy to show the normalize, sort, and compare algorithm to be terminating, sound and complete, and decidability of type checking follows in a straightforward manner.

Theorem 3.3 (Decidability) *It is decidable whether or not $\Delta; \Gamma; c \vdash e : \tau \triangleright c'$ is derivable in LXres.*

We say that a term (together with a clock) is *stuck* if it is not a value and if no rule of the operational semantics applies to it. Type safety requires that no well-typed term can become stuck:

Theorem 3.4 (Type Safety) *If $\epsilon; \epsilon; \overline{n} \vdash e : \tau \triangleright c$ and $(e, n) \mapsto^* (e', n')$ then (e', n') is not stuck.*

This is shown using the usual subject reduction and progress lemmas, augmented to track clocks.

No rule in the operational semantics will allow a program to perform a function call when its clock reads 0, so any program attempting to exceed its time bounds would become stuck. It therefore follows from type safety that a well-typed program cannot exceed its time bound:

Corollary 3.5 (Adherence to Resource Bounds) *If $\epsilon; \epsilon; \overline{n} \vdash e : \tau \triangleright c$ then e executes in at most n clock steps.*

4 Implementation

We have implemented this work within the framework of the Typed Assembly Language (TAL) security architecture [14, 13]. Our typechecker implements TALres, a variant of TALx86 [13] augmented to include the key features of LXres. An example of TALres code appears in Figure 8.

The features of TAL and LXres interact pleasantly: LXres's virtual clocks may conveniently be considered registers, so LXres's clock-tracking mechanism is easily incorporated into TAL's existing register file mechanism. The kind and constructor languages of TAL are easily augmented to incorporate those of LXres, and the constructor refinement operations are accounted for in TAL by new pseudo-instructions similar to TAL's `unpack` instruction.

TALres also supports a few features in addition to those of LXres. One often requires an enforcement type linking a run-time number to a static natural number. Such an enforcement type can easily be built when numbers are represented by Church numerals, but we wish to avoid the inefficiency of Church numerals and use the built-in integers of TAL. TAL includes a singleton integer type to support the compilation of tagged unions; with the addition of constructor-refining variants of the basic arithmetic operations, TALres allows this type to serve as an enforcement type. TALres also provides primitive natural number operations other than addition, in support of richer cost functions. In all, none of the enhancements to TAL to account for resource bounds have substantially enlarged its trusted computing base.

Outside the trusted computing base, we implemented a prototype compiler generating TALres. Our compiler takes source programs written in PopCron (which includes cost function annotations) and compiles them to TALres, thereby certifying them for safety and for resource bounds. The PopCron language resembles Popcorn [13], which in turn is a safe dialect of C. At the level of a C source program it is difficult to predict the exact number of instructions the resulting executable will use, but the number of jumps is more stable, and for this reason we measure running time in terms of backward jumps. (More precisely, we charge a jump when it cannot be determined to be forward.) The discrepancy between function calls in the source and chargeable jumps in the executable (typically two jumps per function call) is easily compensated for by the compiler.

Each function in PopCron must be annotated with a running time expression given in terms of zero or more of the function's arguments using a dependent type notation. For example, the notation `<|0|>` in the following definition of an increment function specifies that this code makes no function calls.

```
int add1 (int x)<|0|> { return x+1; }
```

The compiler automatically adds representation arguments, threads the cost through the function, and provides representations for function calls. The PopCron language does not currently provide a facility for a function type to specify the representation of its result (recall Section 2.2), but we plan to add such a facility in the future.

Continuing the trivial example, the generated TALres code for the function `add1` is :

```
_add1:
LABELTYPE <All[k:Sint s:Ts].{CLK: 1 + k,
  ESP:sptr {CLK:k, EAX:B4, ESP:sptr B4::s}::B4::s}>
        MOV     EAX,[ESP+4]
        INC     EAX
        RETN
```

The type of this label is a precondition for jumping to it. The precondition says that the stack (contained in register `ESP`) must contain the pointer to a return address, followed by a four byte value (type `B4`), followed by the rest of the stack, `s`, which is held abstract. Furthermore, the clock time, in register `CLK`, is also abstract, though it must be at least 1. This extra clock step is added by the compiler to account for the function's return.

Example Space considerations preclude a thorough discussion of PopCron and the compiler, so we illustrate them by example. Figure 6 contains an example of a more substantial source program written in PopCron. This program first defines the tree type (specialized in this case to integers), using recursive unions and structs. Note that the left and right branches of each tree node are immutable; the PopCron type system does not permit any mutable data to contribute to a cost function. The bulk of this example is the function `app_tree` that applies its argument function `f` to every value in the tree.[1] The type given for `f` specifies that, like `add1`, it may make no function calls. This is just for simplicity, like TALres, PopCron does allow functions to be polymorphic over the time taken by a parameter function.

The time annotation for `app_tree` refers to `cost_tree` in calculating the time taken for the tree `t`. The keyword `time` indicates that `cost_tree` is only used in timing annotations, and, as it will never be executed, does not need its own time annotation.

The compiler's TALres output appears in Figures 7 and 8. The only changes to this code for presentation are

```
struct tree_node {
  const tree left;
  const tree right;
  int val;
}
union tree {
  void leaf;
  tree_node node;
}

time cost_tree (tree t) {
  switch t {
  case leaf : return 0;
  case node(n): return (cost_tree_node(n));
  }
}
time cost_tree_node (tree_node n) {
  return (cost_tree (n.left) +
          cost_tree (n.right) + 3);
}

void app_tree (tree t, void f (int x)<|0|>)
  <| cost_tree(t) |> {
    switch t {
    case leaf : return;
    case node(n) :
      f(n.val);
      app_tree(n.left,f);
      app_tree(n.right,f);
      return;
  }
}
```

Figure 6: Example of PopCron code

some constructor simplification (in some cases they have been replaced with equivalent, but smaller versions) and formatting. For example, we define two common types at the bottom of Figure 8: `ftype` is the type of the function argument to `app_tree`, and `ra` is the type of the return address of the function.

Figure 7 contains the definitions of the TALres kinds constructors and types necessary to annotate the compiled code of `app_tree`. The first two lines of this figure define the mutually recursive kinds representing the `tree` union and `tree_node` struct.[2] The next four lines define unfold for these recursive kinds, where `RECCON` and `ANDCON` provide a mutually recursive version of LXres's `pr`. The two time functions, `cost_tree` and `cost_tree_node`, are also defined with these constructs.

Lines 13 through 22 define the parameterized recursive types of trees and tree nodes,[3] where `rec` creates a tuple of mutually recursive types, and then the individual components are projected out in lines 21 and 22. Like $tree(s)(\alpha)$, in Figure 4, `tree_rep` forms a tagged sum (the branches are discriminated by the singleton types `S(1)` and `S(2)`), and within each branch of the sum, a case constructor allows the argument to be refined with a virtual case.

[1] PopCron follows the terminology of C by using `void` to refer to the trivial type; however, TALres follows standard type-theoretic terminology in which `void` refers to the empty type.

[2] Both sums and products in TALres may contain an arbitrary number of fields. Sums are denoted by `+[` *object*, ... `]`, products by `*[` *object*, ... `]`, and unit is the empty product `*[]` .

[3] Product values must be boxed (unless they lie within an outer boxed object), and consequently product types are often indicated to be boxed by a `^` prefix. Additionally, product types contain flags indicating that a field is read-only (`^r`) or read-write (`^rw`) [13]. For example, a boxed pair of mutable integers would have type `^*[B4^rw,B4^rw]`. For brevity, when each field of a sum is boxed, the boxing prefix may be placed on the entire sum.

```
RECKIND <tree_k = +[*[],tree_node_k]>
ANDKIND <tree_node_k = *[tree_k,tree_k]>
RECCON  <unfold_tree : tree_k -!> +[*[],tree_node_k] =                     \
          fn a$15 : +[*[],tree_node_k] . a$15>
ANDCON  <unfold_tree_node : tree_node_k -!> *[tree_k,tree_k] =             \
          fn a$13 : *[tree_k,tree_k] . a$13>

RECCON  <cost_tree : tree_k -!> Nat =                                      \
          fn t : +[*[],tree_node_k] . case (t) beta [0,cost_tree_node beta]>
ANDCON  <cost_tree_node : tree_node_k -!> Nat =                            \
          fn n : *[tree_k,tree_k] . 3 + (cost_tree n.0)+ (cost_tree n.1)>

TYPE    <_reptype = rec(
          tree_rep:tree_k-!>T4. fn union$16:tree_k .                       \
                ^+[*[S(1)^r,case (unfold_tree union$16) b$17 [B4,void[T4]]^r],  \
                   *[S(2)^r,case (unfold_tree union$16) b$18 [void[T4],     \
                               tree_node_rep b$18]^r]],                    \
          tree_node_rep:tree_node_k-!>T4. fn struct$14:tree_node_k .       \
                  ^*[(tree_rep (unfold_tree_node struct$14).0)^r,          \
                     (tree_rep (unfold_tree_node struct$14).1)^r,B4^rw])>
TYPE    <tree_rep      = _reptype.0>
TYPE    <tree_node_rep = _reptype.1>
```

Figure 7: TALres kind and constructor definitions

Figure 8 contains the TALres output of the compiler for `app_tree`. Like `add1`, the precondition of this label expects that the return address and then the arguments (first the tree, then the function) will be passed to it on the stack. Furthermore, the time annotation of `app_tree` has been compiled to the constructor in the `CLK` register. As function calls are composed of two jumps, the PopCron time annotation is doubled, and an extra step is added for the final jump at the function return.

The first step of the code is to explicitly unroll the recursive type, and then branch on the sum tag, using TAL's pseudo-instruction `BTAGVAR`. If the tag is equal[4] to 1, the code jumps to label `leaf_void$21`, otherwise it falls through. At `leaf_void$21`, as `BTAGVAR` has removed all but one branch, `EAX` contains a pointer to the degenerate sum

```
^+[*[S(1)^r,case (unfold_tree t)
            b$17 [B4,void[T4]]^r]]
```

so it is explicitly coerced to remove the sum wrapping. Next, `t` is refined in the next two lines. The `VCASE` instruction[5] examines the type of second component of the product, verifying that it would be void if the argument to the case were `(fold (inj 1 beta))`. After refinement, `t` has been replaced by `(fold (inj 0 beta))` so the clock reads

```
1 + (cost_tree (fold (inj 0 beta)))
  + (cost_tree (fold (inj 0 beta))) + k
```

which normalizes to `1+k`. After the clock is stepped for the jump, this gives the time expected by the return address.

In the other branch, `node_value$22` also coerces the `EAX` from a degenerate sum so that its type becomes

```
^*[S(2)^r,case (unfold_tree t) b$18
          [void[T4], tree_node_rep b$18]^r]
```

After refinement (Lines 8 and 9), this type is equivalent to `^*[S(2)^r, (tree_node_rep n)^r]`, for a new variable `n`. In lines 10 and 11, the second component of this tuple is pushed onto the stack and also put into `EAX`. After more refinement in lines 12 and 13, `EAX` is of type `tree_node_rep [left3,right4]`, for new variables `left3` and `right4`. This recursive type is explicitly unrolled so that the third component can be extracted and placed on the stack in line 15. The argument `f` is then fetched from the stack and called in line 17. Before the call, the clock reads (through the refinements):

```
7 + cost_tree(left3) + cost_tree(left3)
  + cost_tree(right4) + cost_tree(right4) + k
```

As the call to `f` takes one step, and `f` takes one step to return, we expect that after the call the clock should read

```
5 + cost_tree(left3) + cost_tree(left3)
  + cost_tree(right4) + cost_tree(right4) + k
```

and so pass this constructor argument to `f`. The second constructor argument is the current state of the stack minus the argument to `f`.

On return from `f` we pop the argument off the stack (line 21), move the pointer for `f` up the stack to prepare for the recursive call (22), and then extract the left child of the tree (23–24). The call (25) requires the type application of the time after the call, the constructor representing the left child, and the stack. Lines 29–34 perform a similar call with the right child. At Line 36, the clock reads `1+k`, allowing the return.

5 Conclusion

Adherence to resource bounds is an important safety property for untrusted agents in real-world situations. Our type system certifies programs by augmenting them with virtual clocks, and proving that the clocks of well-typed programs cannot expire. This mechanism was first suggested by Necula and Lee [17] for the PCC framework. This work extends theirs by allowing executions to vary in length depending on their input, and by providing a fully automatic compiler generating executables certified for resource bounds.

[4] The "E" in the `BTAGVAR` instruction indicates a test for equality.

[5] The `VCASE` instruction takes four arguments: the number of the live branch, a new constructor variable to bind, the constructor being case analyzed, and the value residing in the dead branch.

```
    _app_tree:
    LABELTYPE <All[k:Nat t:tree_k s:Ts].{CLK: 1+(cost_tree t)+(cost_tree t)+k,       \
      ESP: sptr ra :: (tree_rep t) :: ftype :: s}>
            MOV     EAX,unroll([ESP+4])
            BTAGVAR E,[EAX+0],1,leaf_void$21
    node_value$22:
            COERCE  unisum(EAX)
            LETFOLD alpha1,t
            VCASE   1,n,alpha1,[EAX+4]
            PUSH    DWORD PTR [EAX+4]
            MOV     EAX,[ESP+0]
            LETFOLD r$24,n
            LETPROD [left3,right4],r$24
            COERCE  unroll(EAX)
            PUSH    DWORD PTR [EAX+8]
            MOV     EAX,[ESP+16]
            CALL    tapp(EAX,<5 + (cost_tree left3) + (cost_tree right4) +           \
                                + (cost_tree left3) + (cost_tree right4) + k,        \
                              tree_node_rep [left3,right4] :: ra                     \
                                :: (tree_rep t) :: ftype :: s >)
            ADD     ESP,4
            PUSH    DWORD PTR [ESP+12]
            MOV     EAX,unroll([ESP+4])
            PUSH    DWORD PTR [EAX+0]
            CALL    tapp(_app_tree,<3 + (cost_tree right4) + (cost_tree right4) + k, \
                               left3, tree_node_rep [left3,right4] :: ra             \
                                 :: (tree_rep t) :: ftype :: s >)
            ADD     ESP,8
            PUSH    DWORD PTR [ESP+12]
            MOV     EAX,unroll([ESP+4])
            PUSH    DWORD PTR [EAX+4]
            CALL    tapp(_app_tree,<1 + k,  right4,                                  \
                               tree_node_rep [left3,right4] :: ra                    \
                                 :: (tree_rep t) :: ftype :: s>)
            ADD     ESP,12
            RETN
    leaf_void$21:
            COERCE  unisum(EAX)
            LETFOLD alpha1,t
            VCASE   0,beta2,alpha1,[EAX+4]
            RETN

where ftype =  All[k:Nat s:Ts].{CLK: 1+k, ESP: sptr { CLK: k, ESP: sptr B4::s} :: B4 :: s}
and   ra    =  { CLK: k, ESP: sptr (tree_rep t) :: ftype :: s }
```

Figure 8: Compilation of the function `app_tree`

Like Necula and Lee, we make no effort to *infer* resource bounds, and instead rely on annotations supplied by the programmer.

Some other type systems for controlling resource consumption are Hoffman's linear type system for (asymptotically) bounding time [8] and Hughes, Pareto and Sabry's sized types for bounding space [10, 9]. These type systems provide different expressive power from ours: Hoffman provides asymptotic bounds, and Hughes *et al.* account for heap space, but neither allow bounds to depend on input data, and both work at the level of source code, not executables.

Most similar to our work is Reistad and Gifford's type system for expressing the running time of programs as a function of their inputs [19]; their work differs mainly in that they limit the programmer to what can be done using a set of primitive cost-related constructs, whereas we build such constructs from basic mechanisms, and (like Hoffman and Hughes *et al.*) they operate at the level of source, not executables. However, Reistad and Gifford consider cost inference, which we do not, so it may well be profitable to combine their source language with our executable language.

A key contribution of this work is a programming idiom for simulating dependent types using sum and inductive kinds. This idiom does make programs more complex than they would be in a language that included dependent types; the benefit of our approach lies in its substantially simpler type theory and resulting ease of type checking (*i.e.*, verification). We prefer a simpler type theory because, aside from a general preference for simpler type theories, simplicity is key to the robustness of our system. The type checker is part of the trusted computing base of our security infrastructure, so any complexity there makes the system less likely to be secure. The compiler, on the other hand, is not part of the trusted computing base, so any error there merely leads to rejected code.

The principal limitation of our approach lies in the limitations that exist on cost functions. First, cost functions must be primitive recursive, which rules out some resource bounds. Second, cost functions are limited to using a fixed set of built-in arithmetic operators. As discussed in Section 3.3, defined arithmetic operators often do not work as intended because of the lack of an eta-equivalence rule for inductive kinds. Third, cost functions are limited to using aspects of their metric arguments that can be statically represented (or, more precisely, statically represented in a manner admitting an enforcement type). In TALres many sorts of data are not easily represented, such as cyclic data structures. Despite these limitations, we believe that our approach is applicable to a wide variety of important applications.

This work easily generalizes to some other sorts of resource bounds. Easiest is stack space; by replacing the virtual clock with a record of remaining stack space, we may statically prevent stack overflow. Our type system already accounts for the complications involved in recovering resources (recall Section 2). Our approach also appears to generalize to heap space, but to do so we must make the type system aware of when heap space is reclaimed, which it is not when memory is reclaimed by a garbage collector. We conjecture that this can be done using elements of the typed memory management language of Crary *et al.* [2], which was designed to expose memory management primitives in Typed Assembly Language. Finally, we can account for resource consumption *rates* by having the program intermittently yield and by replenishing its resource allowance when it is next scheduled after a yield.

Acknowledgements

We thank Dan Grossman, Greg Morrisett, David Walker and Steve Zdancewic for helpful comments, and the TAL group at Cornell University for the Popcorn compiler and TALx86 verifier, the basis of our PopCron compiler and TALres verifier.

References

[1] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Dec. 1995.

[2] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, Jan. 1999.

[3] K. Crary and S. Weirich. Flexible type analysis. In *1999 ACM International Conference on Functional Programming*, pages 233–248, Paris, Sept. 1999.

[4] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *1998 ACM International Conference on Functional Programming*, pages 301–312, Baltimore, Sept. 1998. Extended version published as Cornell University technical report TR98-1721.

[5] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

[6] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[7] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, Jan. 1995.

[8] M. Hoffman. Linear types and non-size increasing polynomial time computation. In *Fourteenth IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999.

[9] J. Hughes and L. Pareto. Recursion and dynamic datastructures in bounded space: Towards embedded ML programming. In *1999 ACM International Conference on Functional Programming*, pages 70–81, Paris, Sept. 1999.

[10] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 410–423, St. Petersburg, Florida, Jan. 1996.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[12] N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172, 1991.

[13] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, Atlanta, May 1999.

[14] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 1999. To appear. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

[15] G. Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, Jan. 1997.

[16] G. Necula and P. Lee. Safe kernel extensions without runtime checking. In *Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, Oct. 1996.

[17] G. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In *Special Issue on Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science.* Springer-Verlag, Oct. 1997.

[18] G. Necula and P. Lee. The design and implementation of a certifying compiler. In *1998 SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, June 1998.

[19] B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *1994 ACM Conference on Lisp and Functional Programming*, pages 65–78, Orlando, Florida, June 1994.

[20] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, Dec. 1993.

[21] H. Xi and F. Pfenning. Dependent types in practical programming. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas, Jan. 1999.

A Static semantics

A.1 Kind formation

$\boxed{\Delta \vdash k\ \mathsf{kind}}$

$$\frac{}{\Delta \vdash \mathsf{Type\ kind}} \qquad \frac{}{\Delta \vdash \mathsf{Nat\ kind}} \qquad \frac{}{\Delta \vdash 1\ \mathsf{kind}}$$

$$\frac{}{\Delta \vdash j\ \mathsf{kind}}\ (j \in \Delta) \qquad \frac{\Delta, j \vdash k\ \mathsf{kind}}{\Delta \vdash \mu j.k\ \mathsf{kind}} \begin{pmatrix} j \text{ only positive in } k \\ j \notin \Delta \end{pmatrix}$$

$$\frac{\Delta \vdash k_1\ \mathsf{kind} \qquad \Delta \vdash k_2\ \mathsf{kind}}{\Delta \vdash k_1 \to k_2\ \mathsf{kind}} \qquad \frac{\Delta \vdash k_1\ \mathsf{kind} \qquad \Delta \vdash k_2\ \mathsf{kind}}{\Delta \vdash k_1 + k_2\ \mathsf{kind}}$$

$$\frac{\Delta \vdash k_1\ \mathsf{kind} \qquad \Delta \vdash k_2\ \mathsf{kind}}{\Delta \vdash k_1 \times k_2\ \mathsf{kind}}$$

A.2 Constructor Formation

$\boxed{\Delta \vdash c : k}$

$$\frac{}{\Delta \vdash * : 1} \qquad \frac{}{\Delta \vdash \alpha : \Delta(\alpha)}$$

$$\frac{\Delta, \alpha{:}k' \vdash c : k \qquad \Delta \vdash k' \ \mathsf{kind}}{\Delta \vdash \lambda\alpha{:}k'.c : k' \to k} \ (\alpha \notin \Delta)$$

$$\frac{\Delta \vdash c_1 : k' \to k \qquad \Delta \vdash c_2 : k'}{\Delta \vdash c_1 c_2 : k} \qquad \frac{\Delta \vdash c_1 : k_1 \qquad \Delta \vdash c_2 : k_2}{\Delta \vdash \langle c_1, c_2 \rangle : k_1 \times k_2}$$

$$\frac{\Delta \vdash c : k_1 \times k_2}{\Delta \vdash \mathtt{prj}_1\, c : k_1} \qquad \frac{\Delta \vdash c : k_1 \times k_2}{\Delta \vdash \mathtt{prj}_2\, c : k_2}$$

$$\frac{\Delta \vdash c : k_1 \qquad \Delta \vdash k_2 \ \mathsf{kind}}{\Delta \vdash \mathtt{inj}_1^{k_1+k_2}\, c : k_1 + k_2} \qquad \frac{\Delta \vdash c : k_2 \qquad \Delta \vdash k_1 \ \mathsf{kind}}{\Delta \vdash \mathtt{inj}_2^{k_1+k_2}\, c : k_1 + k_2}$$

$$\frac{\begin{array}{c}\Delta \vdash c : k_1 + k_2 \\ \Delta, \alpha{:}k_1 \vdash c_1 : k \\ \Delta, \alpha{:}k_2 \vdash c_2 : k\end{array}}{\Delta \vdash \mathtt{case}(c, \alpha.c_1, \alpha.c_2) : k} \ (\alpha \notin \Delta) \qquad \frac{\Delta \vdash c : k[\mu j.k/j]}{\Delta \vdash \mathtt{fold}_{\mu j.k}\, c : \mu j.k}$$

$$\frac{\begin{array}{c}\Delta, j, \alpha{:}k, \varphi{:}j \to k' \vdash c : k' \\ \Delta \vdash \mu j.k \ \mathsf{kind} \qquad \Delta, j \vdash k' \ \mathsf{kind}\end{array}}{\Delta \vdash \mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c) : \mu j.k \to k'[\mu j.k/j]}$$
$(j$ only positive in k', and $j, \alpha, \varphi \notin \Delta)$

$$\frac{}{\Delta \vdash \overline{n} : \mathsf{Nat}} \qquad \frac{\Delta \vdash c_1 : \mathsf{Nat} \quad \Delta \vdash c_2 : \mathsf{Nat}}{\Delta \vdash c_1 \dotplus c_2 : \mathsf{Nat}}$$

$$\frac{\Delta, \alpha{:}\mathsf{Nat}, \beta{:}k \vdash c_1 : k \quad \Delta \vdash c_2 : k}{\Delta \vdash \mathtt{prnat}_k(\alpha, \beta.c_1; c_2) : k} \ (\alpha, \beta \notin \Delta)$$

$$\frac{\begin{array}{c}\Delta \vdash \tau_1 : \mathsf{Type} \qquad \Delta \vdash \tau_2 : \mathsf{Type} \\ \Delta \vdash c_1 : \mathsf{Nat} \qquad \Delta \vdash c_2 : \mathsf{Nat}\end{array}}{\Delta \vdash (\tau_1, c_1) \to (\tau_2, c_2) : \mathsf{Type}}$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{Type} \qquad \Delta \vdash \tau_2 : \mathsf{Type}}{\Delta \vdash \tau_1 \times \tau_2 : \mathsf{Type}} \qquad \frac{\Delta \vdash \tau_1 : \mathsf{Type} \qquad \Delta \vdash \tau_2 : \mathsf{Type}}{\Delta \vdash \tau_1 + \tau_2 : \mathsf{Type}}$$

$$\frac{\Delta, \alpha{:}k \vdash \tau : \mathsf{Type} \qquad \Delta \vdash k \ \mathsf{kind}}{\Delta \vdash \forall\alpha{:}k.\tau : \mathsf{Type}} \ (\alpha \notin \Delta)$$

$$\frac{\Delta, \alpha{:}k \vdash \tau : \mathsf{Type} \qquad \Delta \vdash k \ \mathsf{kind}}{\Delta \vdash \exists\alpha{:}k.\tau : \mathsf{Type}} \ (\alpha \notin \Delta)$$

$$\frac{}{\Delta \vdash \mathtt{void} : \mathsf{Type}} \qquad \frac{}{\Delta \vdash \mathtt{unit} : \mathsf{Type}}$$

$$\frac{\begin{array}{c}\Delta \vdash c : (k \to \mathsf{Type}) \to k \to \mathsf{Type} \\ \Delta \vdash k \ \mathsf{kind} \qquad \Delta \vdash c' : k\end{array}}{\Delta \vdash \mathtt{rec}_k(c, c') : \mathsf{Type}}$$

A.3 Constructor Equivalence

$\boxed{\Delta \vdash c = c' : k}$

$$\frac{\begin{array}{c}\Delta \vdash c' : k[\mu j.k/j] \qquad \Delta, j \vdash k' \ \mathsf{kind} \\ \Delta, j, \alpha{:}k, \varphi{:}j \to k' \vdash c : k' \quad \Delta \vdash \mu j.k \ \mathsf{kind}\end{array}}{\begin{array}{l}\Delta \vdash \mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c)(\mathtt{fold}_{\mu j.k}\, c') = \\ \quad c[\mu j.k, c', \mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c)/j, \alpha, \varphi] : k'[\mu j.k/j]\end{array}}$$
$(j$ only positive in k' and $j, \alpha, \varphi \notin \Delta)$

$$\frac{\Delta \vdash c_1 : k \qquad \Delta \vdash c_2 : k'}{\Delta \vdash \mathtt{prj}_1 \langle c_1, c_2 \rangle = c_1 : k}$$

$$\frac{\Delta \vdash c_1 : k' \qquad \Delta \vdash c_2 : k}{\Delta \vdash \mathtt{prj}_2 \langle c_1, c_2 \rangle = c_2 : k}$$

$$\frac{\Delta \vdash c : k_1 \times k_2}{\Delta \vdash \langle \mathtt{prj}_1\, c, \mathtt{prj}_2\, c \rangle = c : k_1 \times k_2}$$

$$\frac{\begin{array}{c}\Delta \vdash k' \ \mathsf{kind} \\ \Delta, \alpha{:}k' \vdash c : k \qquad \Delta \vdash c' : k\end{array}}{\Delta \vdash (\lambda\alpha{:}k'.c)c' = c[c'/\alpha] : k} \ (\alpha \notin \Delta)$$

$$\frac{\Delta \vdash c : k' \to k}{\Delta \vdash (\lambda\alpha{:}k'.c\alpha) = c : k' \to k} \ (\alpha \notin FV(c))$$

$$\frac{\begin{array}{c}\Delta, \alpha{:}k_1 \vdash c_1 : k \qquad \Delta, \alpha{:}k_2 \vdash c_2 : k \\ \Delta \vdash c : k_1 \qquad \Delta \vdash k_2 \ \mathsf{kind}\end{array}}{\Delta \vdash \mathtt{case}(\mathtt{inj}_1^{k_1+k_2}\, c, \alpha.c_1, \alpha.c_2) = c_1[c/\alpha] : k}$$

$$\frac{\begin{array}{c}\Delta, \alpha{:}k_1 \vdash c_1 : k \qquad \Delta, \alpha{:}k_2 \vdash c_2 : k \\ \Delta \vdash c : k_2 \qquad \Delta \vdash k_1 \ \mathsf{kind}\end{array}}{\Delta \vdash \mathtt{case}(\mathtt{inj}_2^{k_1+k_2}\, c, \alpha.c_1, \alpha.c_2) = c_2[c/\alpha] : k}$$

$$\frac{\Delta \vdash c : k_1 + k_2}{\begin{array}{l}\Delta \vdash \mathtt{case}(c, \alpha_1.\, \mathtt{inj}_1^{k_1+k_2}\, \alpha_1, \alpha_2.\, \mathtt{inj}_2^{k_1+k_2}\, \alpha_2) = \\ \quad c : k_1 + k_2\end{array}}$$

$$\frac{\Delta \vdash c_1 : \mathsf{Nat} \quad \Delta \vdash c_2 : \mathsf{Nat}}{\Delta \vdash c_1 \dotplus c_2 = c_2 \dotplus c_1 : \mathsf{Nat}}$$

$$\frac{\Delta \vdash c_1 : \mathsf{Nat} \quad \Delta \vdash c_2 : \mathsf{Nat} \quad \Delta \vdash c_3 : \mathsf{Nat}}{\Delta \vdash c_1 \dotplus (c_2 \dotplus c_3) = (c_1 \dotplus c_2) \dotplus c_3 : \mathsf{Nat}}$$

$$\frac{\Delta \vdash c : \mathsf{Nat}}{\Delta \vdash c \dotplus \overline{0} = c : \mathsf{Nat}}$$

$$\frac{}{\Delta \vdash \overline{n_1} \dotplus \overline{n_2} = \overline{n_1 + n_2} : \mathsf{Nat}}$$

$$\frac{\Delta, \alpha{:}\mathsf{Nat}, \beta{:}k \vdash c_1 : k \quad \Delta \vdash c_2 : k}{\Delta \vdash \mathtt{prnat}_k(\alpha, \beta.c_1; c_2)\, \overline{0} = c_2 : k} \ (\alpha, \beta \notin \Delta)$$

$$\frac{\Delta, \alpha{:}\mathsf{Nat}, \beta{:}k \vdash c_1 : k \quad \Delta \vdash c_2 : k \quad \Delta \vdash c : \mathsf{Nat}}{\begin{array}{l}\Delta \vdash \mathtt{prnat}_k(\alpha, \beta.c_1; c_2)(c \dotplus \overline{1}) = \\ \quad c_1[c, \mathtt{prnat}_k(\alpha, \beta.c_1; c_2)(c)/\alpha, \beta] : k\end{array}} \ (\alpha, \beta \notin \Delta)$$

$$\frac{\Delta \vdash c : k}{\Delta \vdash c = c : k} \qquad \frac{\Delta \vdash c' = c : k}{\Delta \vdash c = c' : k}$$

$$\frac{\Delta \vdash c_1 = c_2 : k \qquad \Delta \vdash c_2 = c_3 : k}{\Delta \vdash c_1 = c_3 : k}$$

$$\frac{\Delta, \alpha{:}k' \vdash c = c' : k \qquad \Delta \vdash k' \ \mathsf{kind}}{\Delta \vdash \lambda\alpha{:}k'.c = \lambda\alpha{:}k'.c' : k' \to k} \ (\alpha \notin \Delta)$$

$$\frac{\Delta \vdash c_1 = c_1' : k' \to k \qquad \Delta \vdash c_2 = c_2' : k'}{\Delta \vdash c_1 c_2 = c_1' c_2' : k}$$

$$\frac{\Delta \vdash c_1 = c_1' : k_1 \qquad \Delta \vdash c_2 = c_2' : k_2}{\Delta \vdash \langle c_1, c_2 \rangle = \langle c_1', c_2' \rangle : k_1 \times c_2}$$

$$\frac{\Delta \vdash c = c' : k_1 \times k_2}{\Delta \vdash \mathtt{prj}_1\, c = \mathtt{prj}_1\, c' : k_1}$$

$$\frac{\Delta \vdash c = c' : k_1 \times k_2}{\Delta \vdash \mathtt{prj}_2\, c = \mathtt{prj}_2\, c' : k_2}$$

$$\frac{\Delta \vdash c = c' : k_1 \qquad \Delta \vdash k_2\ \mathsf{kind}}{\Delta \vdash \mathtt{inj}_1^{k_1+k_2}\, c = \mathtt{inj}_1^{k_1+k_2}\, c' : k_1 + k_2}$$

$$\frac{\Delta \vdash c = c' : k_2 \qquad \Delta \vdash k_1\ \mathsf{kind}}{\Delta \vdash \mathtt{inj}_2^{k_1+k_2}\, c = \mathtt{inj}_2^{k_1+k_2}\, c' : k_1 + k_2}$$

$$\frac{\begin{array}{c}\Delta \vdash c = c' : k_1 + k_2 \\ \Delta, \alpha{:}k_1 \vdash c_1 = c_1' : k \\ \Delta, \alpha{:}k_2 \vdash c_2 = c_2' : k\end{array}}{\begin{array}{l}\Delta \vdash \mathtt{case}(c, \alpha.c_1, \alpha.c_2) = \\ \quad \mathtt{case}(c', \alpha.c_1', \alpha.c_2') : k\end{array}}\ (\alpha \notin \Delta)$$

$$\frac{\Delta \vdash c = c' : k[\mu j.k/j]}{\Delta \vdash \mathtt{fold}_{\mu j.k}\, c = \mathtt{fold}_{\mu j.k}\, c' : \mu j.k}$$

$$\frac{\begin{array}{c}\Delta, j, \alpha{:}k, \varphi{:}j \to k' \vdash c_1 = c_2 : k' \\ \Delta \vdash \mu j.k\ \mathsf{kind} \qquad \Delta, j \vdash k'\ \mathsf{kind}\end{array}}{\begin{array}{l}\Delta \vdash \mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c_1) = \mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c_2) \\ \quad : \mu j.k \to k'[\mu j.k/j]\end{array}}$$
$$(j \text{ only positive in } k' \text{ and } j, \alpha, \varphi \notin \Delta)$$

$$\frac{\Delta \vdash c_1 = c_1' : \mathsf{Nat} \quad \Delta \vdash c_2 = c_2' : \mathsf{Nat}}{\Delta \vdash c_1 \dotplus c_2 = c_1' \dotplus c_2' : \mathsf{Nat}}$$

$$\frac{\Delta, \alpha{:}\mathsf{Nat}, \beta{:}k \vdash c_1 = c_1' : k \quad \Delta \vdash c_2 = c_2' : k}{\Delta \vdash \mathtt{prnat}_k(\alpha, \beta.c_1; c_2) = \mathtt{prnat}_k(\alpha, \beta.c_1'; c_2') : k}\ (\alpha, \beta \notin \Delta)$$

$$\frac{\begin{array}{c}\Delta \vdash \tau_1 = \tau_1' : \mathsf{Type} \qquad \Delta \vdash \tau_2 = \tau_2' : \mathsf{Type} \\ \Delta \vdash c_1 = c_1' : \mathsf{Nat} \qquad \Delta \vdash c_2 = c_2' : \mathsf{Nat}\end{array}}{\Delta \vdash (\tau_1, c_1) \to (\tau_2, c_2) = (\tau_1', c_1') \to (\tau_2', c_2') : \mathsf{Type}}$$

$$\frac{\Delta \vdash \tau_1 = \tau_1' : \mathsf{Type} \qquad \Delta \vdash \tau_2 = \tau_2' : \mathsf{Type}}{\Delta \vdash \tau_1 \times \tau_2 = \tau_1' \times \tau_2' : \mathsf{Type}}$$

$$\frac{\Delta \vdash \tau_1 = \tau_1' : \mathsf{Type} \qquad \Delta \vdash \tau_2 = \tau_2' : \mathsf{Type}}{\Delta \vdash \tau_1 + \tau_2 = \tau_1' + \tau_2' : \mathsf{Type}}$$

$$\frac{\Delta, \alpha{:}k \vdash \tau = \tau' : \mathsf{Type} \qquad \Delta \vdash k\ \mathsf{kind}}{\Delta \vdash \forall \alpha{:}k.\tau = \forall \alpha{:}k.\tau' : \mathsf{Type}}\ (\alpha \notin \Delta)$$

$$\frac{\Delta, \alpha{:}k \vdash \tau = \tau' : \mathsf{Type} \qquad \Delta \vdash k\ \mathsf{kind}}{\Delta \vdash \exists \alpha{:}k.\tau = \exists \alpha{:}k.\tau' : \mathsf{Type}}\ (\alpha \notin \Delta)$$

$$\frac{\begin{array}{c}\Delta \vdash k\ \mathsf{kind} \qquad \Delta \vdash c_2 = c_2' : k \\ \Delta \vdash c_1 = c_1' : (k \to \mathsf{Type}) \to k \to \mathsf{Type}\end{array}}{\Delta \vdash \mathtt{rec}_k(c_1, c_2) = \mathtt{rec}_k(c_1', c_2') : \mathsf{Type}}$$

A.4 Term Formation

$\boxed{\Delta; \Gamma; c \vdash e : \tau}$

$$\frac{}{\Delta; \Gamma; c \vdash * : \mathtt{unit} \triangleright c} \qquad \frac{}{\Delta; \Gamma; c \vdash x : \Gamma(x) \triangleright c}$$

$$\frac{\begin{array}{c}\Delta; (\Gamma, x{:}\tau_1); c_1 \vdash e : \tau_2 \triangleright c_2 \\ \Delta \vdash \tau_1 : \mathsf{Type} \qquad \Delta \vdash c_1 : \mathsf{Nat}\end{array}}{\Delta; \Gamma; c \vdash \lambda(x{:}\tau_1, c_1).e : (\tau_1, c_1) \to (\tau_2, c_2) \triangleright c}\ (x \notin \Gamma)$$

$$\frac{\begin{array}{c}\Delta; \Gamma; c \vdash e_1 : (\tau_1, c_1) \to (\tau_2, c_2) \triangleright c' \\ \Delta; \Gamma; c' \vdash e_2 : \tau_1 \triangleright (c_1 \dotplus 1)\end{array}}{\Delta; \Gamma; c \vdash e_1 e_2 : \tau_2 \triangleright c_2}$$

$$\frac{\Delta; \Gamma; c \vdash e : \tau \triangleright (c_1 \dotplus c_2)}{\Delta; \Gamma; c \vdash \mathtt{waste}[c_1]e : \tau \triangleright c_2}$$

$$\frac{\Delta; \Gamma; c \vdash e_1 : \tau_1 \triangleright c_1 \qquad \Delta; \Gamma; c_1 \vdash e_2 : \tau_2 \triangleright c_2}{\Delta; \Gamma; c \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \triangleright c_2}$$

$$\frac{\Delta; \Gamma; c \vdash e : \tau_1 \times \tau_2 \triangleright c'}{\Delta; \Gamma; c \vdash \mathtt{prj}_1\, e : \tau_1 \triangleright c'} \qquad \frac{\Delta; \Gamma; c \vdash e : \tau_1 \times \tau_2 \triangleright c'}{\Delta; \Gamma; c \vdash \mathtt{prj}_2\, e : \tau_2 \triangleright c'}$$

$$\frac{\Delta; \Gamma; c \vdash e : \tau_1 \triangleright c' \qquad \Delta \vdash \tau_2 : \mathsf{Type}}{\Delta; \Gamma; c \vdash \mathtt{inj}_1^{\tau_1+\tau_2}\, e : \tau_1 + \tau_2 \triangleright c'}$$

$$\frac{\Delta; \Gamma; c \vdash e : \tau_2 \triangleright c' \qquad \Delta \vdash \tau_1 : \mathsf{Type}}{\Delta; \Gamma; c \vdash \mathtt{inj}_2^{\tau_1+\tau_2}\, e : \tau_1 + \tau_2 \triangleright c'}$$

$$\frac{\begin{array}{c}\Delta; \Gamma; c \vdash e : \tau_1 + \tau_2 \triangleright c' \\ \Delta; (\Gamma, x{:}\tau_1); c' \vdash e_1 : \tau \triangleright c'' \\ \Delta; (\Gamma, x{:}\tau_2); c' \vdash e_2 : \tau \triangleright c''\end{array}}{\Delta; \Gamma; c \vdash \mathtt{case}(e, x.e_1, x.e_2) : \tau \triangleright c''}\ (x \notin \Gamma)$$

$$\frac{(\Delta, \alpha{:}k); \Gamma; c \vdash v : \tau \triangleright c \qquad \Delta \vdash k\ \mathsf{kind}}{\Delta; \Gamma; c \vdash \Lambda \alpha{:}k.v : \forall \alpha{:}k.\tau \triangleright c}\ (\alpha \notin \Delta)$$

$$\frac{\Delta; \Gamma; c_1 \vdash e : \forall \alpha{:}k.\tau \triangleright c_2 \qquad \Delta \vdash c : k}{\Delta; \Gamma; c_1 \vdash e[c] : \tau[c'/\alpha] \triangleright c_2}$$

$$\frac{\Delta; (\Gamma, f{:}\tau); c \vdash v : \tau \triangleright c \qquad \Delta \vdash \tau : \mathsf{Type}}{\Delta; \Gamma; c \vdash \mathtt{fix}\, f{:}\tau.v : \tau \triangleright c}$$
$$(f \notin \Gamma \text{ and } v = \Lambda \alpha_1{:}k_1 \ldots \Lambda \alpha_n{:}k_n.\lambda(x{:}\tau', c').e')$$

$$\frac{\begin{array}{c}\Delta, \alpha{:}k \vdash \tau : \mathsf{Type} \\ \Delta \vdash c : k \qquad \Delta; \Gamma; c_1 \vdash e : \tau[c/\alpha] \triangleright c_2\end{array}}{\Delta; \Gamma; c_1 \vdash \mathtt{pack}\, e\, \mathtt{as}\, \exists \alpha{:}k.\tau\, \mathtt{hiding}\, c : \exists \alpha{:}k.\tau \triangleright c_2}\ (\alpha \notin \Delta)$$

$$\frac{\begin{array}{c}\Delta; \Gamma; c \vdash e_1 : \exists \alpha{:}k.\tau_2 \triangleright c' \\ (\Delta, \alpha{:}k); (\Gamma, x{:}\tau_2); c' \vdash e_2 : \tau_1 \triangleright c''\end{array}}{\Delta; \Gamma; c \vdash \mathtt{unpack}\, \langle \alpha, x \rangle = e_1\, \mathtt{in}\, e_2 : \tau_1 \triangleright c''}\ \left(\begin{array}{l}\alpha \notin \Delta, FV(\tau) \\ x \notin \Gamma\end{array}\right)$$

$$\frac{\Delta; \Gamma; c_1 \vdash e : \mathtt{rec}_k(c, c') \triangleright c_2}{\Delta; \Gamma; c_1 \vdash \mathtt{unfold}\, e : c(\lambda \alpha{:}k.\mathtt{rec}_k(c, \alpha))c' \triangleright c_2}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;c_1 \vdash e : c(\lambda\alpha{:}k.\,\mathtt{rec}_k(c,\alpha))c' \triangleright c_2 \\ \Delta \vdash \mathtt{rec}_k(c,c') : \mathsf{Type}\end{array}}{\Delta;\Gamma;c_1 \vdash \mathtt{fold}_{\mathtt{rec}_k(c,c')}\, e : \mathtt{rec}_k(c,c') \triangleright c_2}$$

$$\frac{\begin{array}{l}\Delta,\beta{:}k_1,\Delta';\Gamma[\mathtt{inj}_1^{k_1+k_2}\,\beta/\alpha];c_1[\mathtt{inj}_1^{k_1+k_2}\,\beta/\alpha] \vdash \\ \quad v[\mathtt{inj}_1^{k_1+k_2}\,\beta/\alpha] : \mathtt{void} \triangleright c_2' \\ \Delta,\beta{:}k_2,\Delta';\Gamma[\mathtt{inj}_2^{k_1+k_2}\,\beta/\alpha];c_1[\mathtt{inj}_2^{k_1+k_2}\,\beta/\alpha] \vdash \\ \quad e[\mathtt{inj}_2^{k_1+k_2}\,\beta/\alpha] : \tau[\mathtt{inj}_2^{k_1+k_2}\,\beta/\alpha] \triangleright c_2[\mathtt{inj}_2^{k_1+k_2}\,\beta/\alpha] \\ \Delta,\alpha{:}k_1+k_2,\Delta' \vdash c = \alpha : k_1+k_2\end{array}}{\Delta,\alpha{:}k_1+k_2,\Delta';\Gamma;c_1 \vdash \mathtt{vcase}_{\tau\triangleright c_2}(c,\beta.\,\mathtt{dead}\,v,\beta.e) : \tau \triangleright c_2}\ (\beta \notin \Delta)$$

$$\frac{\begin{array}{l}\Delta,\beta{:}k_1,\Delta';\Gamma[\mathtt{inj}_1^{k_1+k_2}\,\beta/\alpha];c_1[\mathtt{inj}_1^{k_1+k_2}\,\beta/\alpha] \vdash \\ \quad e[\mathtt{inj}_1^{k_1+k_2}\,\beta/\alpha] : \tau[\mathtt{inj}_1^{k_1+k_2}\,\beta/\alpha] \triangleright c_2[\mathtt{inj}_1^{k_1+k_2}\,\beta/\alpha] \\ \Delta,\beta{:}k_2,\Delta';\Gamma[\mathtt{inj}_2^{k_1+k_2}\,\beta/\alpha];c_1[\mathtt{inj}_2^{k_1+k_2}\,\beta/\alpha] \vdash \\ \quad v[\mathtt{inj}_2^{k_1+k_2}\,\beta/\alpha] : \mathtt{void} \triangleright c_2' \\ \Delta,\alpha{:}k_1+k_2,\Delta' \vdash c = \alpha : k_1+k_2\end{array}}{\Delta,\alpha{:}k_1+k_2,\Delta';\Gamma;c_1 \vdash \mathtt{vcase}_{\tau\triangleright c_2}(c,\beta.e,\beta.\,\mathtt{dead}\,v) : \tau \triangleright c_2}\ (\beta \notin \Delta)$$

$$\frac{\begin{array}{l}\Delta,\beta{:}k_1,\gamma{:}k_2,\Delta';\Gamma[\langle\beta,\gamma\rangle/\alpha];c_1[\langle\beta,\gamma\rangle/\alpha] \vdash \\ \quad e[\langle\beta,\gamma\rangle/\alpha] : \tau[\langle\beta,\gamma\rangle/\alpha] \triangleright c_2[\langle\beta,\gamma\rangle/\alpha] \\ \Delta,\alpha{:}k_1\times k_2,\Delta' \vdash c = \alpha : k_1 \times k_2\end{array}}{\Delta,\alpha{:}k_1\times k_2,\Delta';\Gamma;c_1 \vdash \mathtt{let}_{\tau\triangleright c_2}\langle\beta,\gamma\rangle = c\,\mathtt{in}\,e : \tau \triangleright c_2}\ (\beta,\gamma \notin \Delta)$$

$$\frac{\begin{array}{l}\Delta,\beta{:}k[\mu j.k/j],\Delta';\Gamma[\mathtt{fold}_{\mu j.k}\,\beta/\alpha] \vdash c_1[\mathtt{fold}_{\mu j.k}\,\beta/\alpha] \vdash \\ \quad e[\mathtt{fold}_{\mu j.k}\,\beta/\alpha] : \tau[\mathtt{fold}_{\mu j.k}\,\beta/\alpha] \triangleright c_2[\mathtt{fold}_{\mu j.k}\,\beta/\alpha] \\ \Delta,\alpha{:}\mu j.k,\Delta' \vdash c = \alpha : \mu j.k\end{array}}{\Delta,\alpha,\Delta'{:}\mu j.k;\Gamma;c_1 \vdash \mathtt{let}_{\tau\triangleright c_2}(\mathtt{fold}_{\mu j.k}\,\beta) = c\,\mathtt{in}\,e : \tau \triangleright c_2}\ (\beta \notin \Delta)$$

$$\frac{\Delta \vdash c = \mathtt{inj}_1^{k_1+k_2}\,c' : k_1+k_2 \qquad \Delta;\Gamma;c_1 \vdash e_1[c'/\alpha] : \tau \triangleright c_2}{\Delta;\Gamma;c_1 \vdash \mathtt{vcase}_{\tau\triangleright c_2}(c,\alpha.e_1,\alpha.\,\mathtt{dead}\,v) : \tau \triangleright c_2}$$

$$\frac{\Delta \vdash c = \mathtt{inj}_2^{k_1+k_2}\,c' : k_1+k_2 \qquad \Delta;\Gamma;c_1 \vdash e_2[c'/\alpha] : \tau \triangleright c_2}{\Delta;\Gamma;c_1 \vdash \mathtt{vcase}_{\tau\triangleright c_2}(c,\alpha.\,\mathtt{dead}\,v,\alpha.e_2) : \tau \triangleright c_2}$$

$$\frac{\Delta \vdash c = \langle c',c''\rangle : k_1\times k_2 \quad \Delta;\Gamma;c_1 \vdash e[c',c''/\beta,\gamma] : \tau \triangleright c_2}{\Delta;\Gamma;c_1 \vdash \mathtt{let}_{\tau\triangleright c_2}\,\langle\beta,\gamma\rangle = c\,\mathtt{in}\,e : \tau \triangleright c_2}$$

$$\frac{\Delta \vdash c = \mathtt{fold}_{\mu j.k}(c') \quad \Delta;\Gamma;c_1 \vdash e[c'/\beta] : \tau \triangleright c_2}{\Delta;\Gamma;c_1 \vdash \mathtt{let}_{\tau\triangleright c_2}\,(\mathtt{fold}_{\mu j.k}\,\beta) = c\,\mathtt{in}\,e : \tau \triangleright c_2}$$

$$\frac{\Delta;\Gamma;c_1 \vdash e : \tau' \triangleright c_2' \quad \Delta \vdash \tau = \tau' : \mathsf{Type} \quad \Delta \vdash c_2 = c_2' : \mathsf{Nat}}{\Delta;\Gamma;c_1 \vdash e : \tau \triangleright c_2}$$

B Operational semantics

Value syntax

$$\begin{array}{rl} v & ::= * \mid \lambda(x{:}\tau,c).e \mid \langle v_1,v_2\rangle \mid \mathtt{inj}_1^{\tau_1+\tau_2}\,v \mid \mathtt{inj}_2^{\tau_1+\tau_2}\,v \\ & \mid \Lambda\alpha{:}k.v \mid (\mathtt{fix}\,f{:}\tau.v)[c_1]\cdots[c_n] \mid \mathtt{fold}_{\mathtt{rec}_k(c,c')}\,v \\ & \mid \mathtt{pack}\,v\,\mathtt{as}\,\exists\alpha.c_1\ \mathtt{hiding}\ c_2 \\ & \mid x \mid \mathtt{prj}_1\,v \mid \mathtt{prj}_2\,v \end{array}$$

Evaluation rules

$$((\lambda(x{:}\tau,c).e)v,n+1) \mapsto (e[v/x],n) \qquad (\text{provided } n \geq 0)$$

$$\frac{(e_1,n)\mapsto(e_1',n')}{(e_1e_2,n)\mapsto(e_1'e_2,n')} \qquad \frac{(e_2,n)\mapsto(e_2',n')}{(ve_2,n)\mapsto(ve_2',n')}$$

$$\frac{(e,n)\mapsto(e',n')}{(\mathtt{waste}[c]e,n)\mapsto(\mathtt{waste}[c]e',n')}$$

$$\frac{c \text{ normalizes to } \overline{n}}{(\mathtt{waste}[c]v,n+n')\mapsto(v,n')} \qquad (\text{provided } n' \geq 0)$$

$$(\mathtt{prj}_1\langle v_1,v_2\rangle,n)\mapsto(v_1,n) \qquad (\mathtt{prj}_2\langle v_1,v_2\rangle,n)\mapsto(v_2,n)$$

$$\frac{(e,n)\mapsto(e',n')}{(\mathtt{prj}_1\,e,n)\mapsto(\mathtt{prj}_1\,e',n')} \qquad \frac{(e,n)\mapsto(e',n')}{(\mathtt{prj}_2\,e,n)\mapsto(\mathtt{prj}_2\,e',n')}$$

$$\frac{(e_1,n)\mapsto(e_1',n')}{(\langle e_1,e_2\rangle,n)\mapsto(\langle e_1',e_2\rangle,n')} \qquad \frac{(e_2,n)\mapsto(e_2',n')}{(\langle v,e_2\rangle,n)\mapsto(\langle v,e_2'\rangle,n')}$$

$$(\mathtt{case}(\mathtt{inj}_1^{\tau_1+\tau_2}\,v,x_1.e_1,x_2.e_2),n)\mapsto(e_1[v/x_1],n)$$

$$(\mathtt{case}(\mathtt{inj}_2^{\tau_1+\tau_2}\,v,x_1.e_1,x_2.e_2),n)\mapsto(e_2[v/x_2],n)$$

$$\frac{(e,n)\mapsto(e',n')}{(\mathtt{inj}_1^{\tau_1+\tau_2}\,e,n)\mapsto(\mathtt{inj}_1^{\tau_1+\tau_2}\,e',n')}$$

$$\frac{(e,n)\mapsto(e',n')}{(\mathtt{inj}_2^{\tau_1+\tau_2}\,e,n)\mapsto(\mathtt{inj}_2^{\tau_1+\tau_2}\,e',n')}$$

$$\frac{(e,n)\mapsto(e',n')}{(\mathtt{case}(e,x_1.e_1,x_2.e_2),n)\mapsto(\mathtt{case}(e',x_1.e_1,x_2.e_2),n')}$$

$$(\Lambda\alpha{:}k.v[c],n)\mapsto(v[c/\alpha],n) \qquad \frac{(e,n)\mapsto(e',n')}{(e[c],n)\mapsto(e'[c],n')}$$

$$\frac{c \text{ normalizes to } \mathtt{inj}_1\,c'}{(\mathtt{vcase}(c,\alpha_1.e_1,\alpha_2.\,\mathtt{dead}\,v),n)\mapsto(e_1[c'/\alpha_1],n)}$$

$$\frac{c \text{ normalizes to } \mathtt{inj}_2\,c'}{(\mathtt{vcase}(c,\alpha_1.\,\mathtt{dead}\,v,\alpha_2.e_2),n)\mapsto(e_2[c'/\alpha_2],n)}$$

$$\frac{c \text{ normalizes to } \langle c_1,c_2\rangle}{(\mathtt{let}\,\langle\beta,\gamma\rangle = c\,\mathtt{in}\,e,n)\mapsto(e[c_1,c_2/\beta,\gamma],n)}$$

$$\frac{c \text{ normalizes to } \mathtt{fold}_{\mu j.k}\,c'}{(\mathtt{let}\,(\mathtt{fold}_{\mu j.k}\,\beta) = c\,\mathtt{in}\,e,n)\mapsto(e[c'/\beta],n)}$$

$$((\mathtt{fix}\,f{:}\tau.v)[c_1]\cdots[c_n]v',n)\mapsto((v[\mathtt{fix}\,f{:}\tau.v/f])[c_1]\cdots[c_n]v',n)$$

$$\frac{(e,n)\mapsto(e',n')}{(\mathtt{pack}\,e\,\mathtt{as}\,\exists\beta.c_1\ \mathtt{hiding}\ c_2,n)\mapsto(\mathtt{pack}\,e'\,\mathtt{as}\,\exists\beta.c_1\ \mathtt{hiding}\ c_2,n')}$$

$$(\mathtt{unpack}\,\langle\alpha,x\rangle = (\mathtt{pack}\,v\,\mathtt{as}\,\tau\ \mathtt{hiding}\ c)\,\mathtt{in}\,e,n)\mapsto(e[c,v/\alpha,x],n)$$

$$\frac{(e,n)\mapsto(e',n')}{(\mathtt{unpack}\,\langle\alpha,x\rangle = e\,\mathtt{in}\,e_2,n)\mapsto(\mathtt{unpack}\,\langle\alpha,x\rangle = e'\,\mathtt{in}\,e_2,n')}$$

$$(\mathtt{unfold}\,(\mathtt{fold}_{\mathtt{rec}_k(c,c')}\,v),n)\mapsto(v,n)$$

$$\frac{(e,n)\mapsto(e',n')}{(\mathtt{fold}_{\mathtt{rec}_k(c,c')}\,e,n)\mapsto(\mathtt{fold}_{\mathtt{rec}_k(c,c')}\,e',n')}$$

$$\frac{(e,n)\mapsto(e',n')}{(\mathtt{unfold}\,e,n)\mapsto(\mathtt{unfold}\,e',n')}$$