Abstractions to control the future

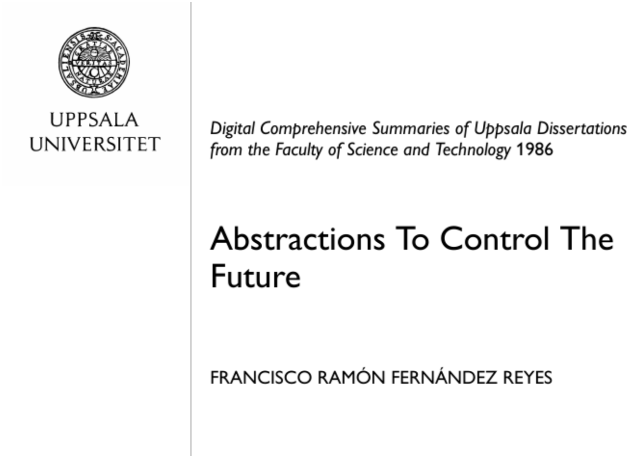
Defense of Francisco Ramón (Kiko) Fernández Reyes

Martin Steffen

18th. January 2021

What’s the field?

What's the field?



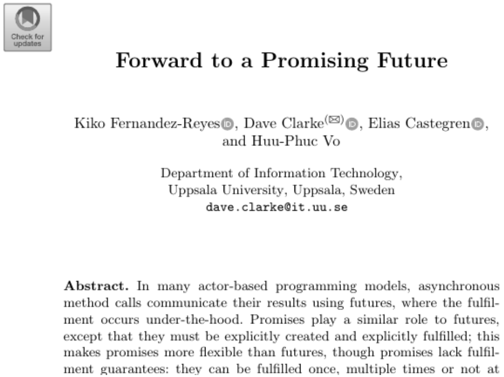
UPPSALA UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1986

Abstractions To Control The Future

FRANCISCO RAMÓN FERNÁNDEZ REYES

What’s the field?



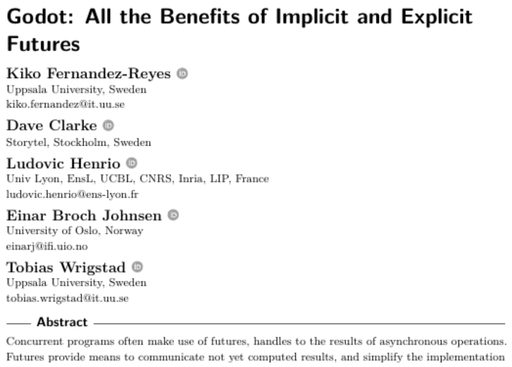
Forward to a Promising Future

Kiko Fernandez-Reyes, Dave Clarke[(✉)], Elias Castegren, and Huu-Phuc Vo

Department of Information Technology, Uppsala University, Uppsala, Sweden
dave.clarke@it.uu.se

Abstract. In many actor-based programming models, asynchronous method calls communicate their results using futures, where the fulfilment occurs under-the-hood. Promises play a similar role to futures, except that they must be explicitly created and explicitly fulfilled; this makes promises more flexible than futures, though promises lack fulfilment guarantees: they can be fulfilled once, multiple times or not at

What's the field?



Godot: All the Benefits of Implicit and Explicit Futures

Kiko Fernandez-Reyes
Uppsala University, Sweden
kiko.fernandez@it.uu.se

Dave Clarke
Storytel, Stockholm, Sweden

Ludovic Henrio
Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, France
ludovic.henrio@ens-lyon.fr

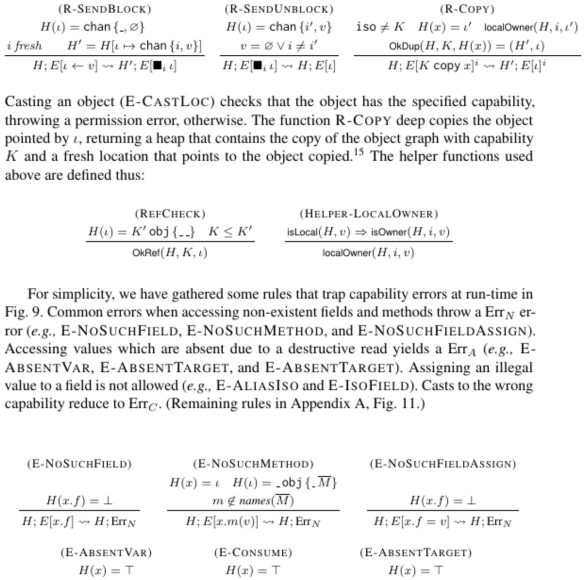
Einar Broch Johnsen
University of Oslo, Norway
einarj@ifi.uio.no

Tobias Wrigstad
Uppsala University, Sweden
tobias.wrigstad@it.uu.se

Abstract

Concurrent programs often make use of futures, handles to the results of asynchronous operations. Futures provide means to communicate not yet computed results, and simplify the implementation

What's the field?



$$\frac{H(\iota) = \mathsf{chan}\,\{\_, \varnothing\} \qquad i\ \mathit{fresh} \qquad H' = H[\iota \mapsto \mathsf{chan}\,\{i, v\}]}{H; E[\iota \leftarrow v] \rightsquigarrow H'; E[\blacksquare_i\, \iota]}\ (\text{R-SendBlock})$$

$$\frac{H(\iota) = \mathsf{chan}\,\{i', v\} \qquad v = \varnothing \lor i \neq i'}{H; E[\blacksquare_i\, \iota] \rightsquigarrow H; E[\iota]}\ (\text{R-SendUnblock})$$

$$\frac{\mathsf{iso} \neq K \qquad H(x) = \iota' \qquad \mathsf{localOwner}(H, i, \iota') \qquad \mathsf{OkDup}(H, K, H(x)) = (H', \iota)}{H; E[K\ \mathsf{copy}\ x]^i \rightsquigarrow H'; E[\iota]^i}\ (\text{R-Copy})$$

Casting an object (E-CastLoc) checks that the object has the specified capability, throwing a permission error, otherwise. The function R-Copy deep copies the object pointed by ι, returning a heap that contains the copy of the object graph with capability K and a fresh location that points to the object copied.[15] The helper functions used above are defined thus:

$$\frac{H(\iota) = K'\ \mathsf{obj}\,\{\_\_\} \qquad K \leq K'}{\mathsf{OkRef}(H, K, \iota)}\ (\text{RefCheck}) \qquad \frac{\mathsf{isLocal}(H, v) \Rightarrow \mathsf{isOwner}(H, i, v)}{\mathsf{localOwner}(H, i, v)}\ (\text{Helper-LocalOwner})$$

For simplicity, we have gathered some rules that trap capability errors at run-time in Fig. 9. Common errors when accessing non-existent fields and methods throw a Err_N error (*e.g.*, E-NoSuchField, E-NoSuchMethod, and E-NoSuchFieldAssign). Accessing values which are absent due to a destructive read yields a Err_A (*e.g.*, E-AbsentVar, E-AbsentTarget, and E-AbsentTarget). Assigning an illegal value to a field is not allowed (*e.g.*, E-AliasIso and E-IsoField). Casts to the wrong capability reduce to Err_C. (Remaining rules in Appendix A, Fig. 11.)

$$\frac{H(x.f) = \bot}{H; E[x.f] \rightsquigarrow H; \mathrm{Err}_N}\ (\text{E-NoSuchField}) \qquad \frac{H(x) = \iota \qquad H(\iota) = \_\mathsf{obj}\,\{\_\,\overline{M}\} \qquad m \notin \mathit{names}(\overline{M})}{H; E[x.m(v)] \rightsquigarrow H; \mathrm{Err}_N}\ (\text{E-NoSuchMethod}) \qquad \frac{H(x.f) = \bot}{H; E[x.f = v] \rightsquigarrow H; \mathrm{Err}_N}\ (\text{E-NoSuchFieldAssign})$$

(E-AbsentVar) $H(x) = \top$ (E-Consume) $H(x) = \top$ (E-AbsentTarget) $H(x) = \top$

Language design

Language design

Abstractions

Concurrency

Sequential

6. Piano.

Concurrency

Sequential

6. Piano.

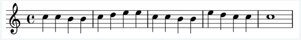
Concurrent

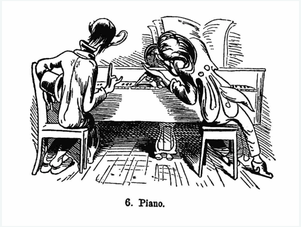
Rallentando
Lent
ppp
8va

14. Finale furioso.

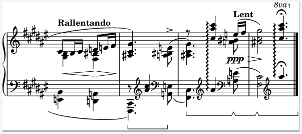
Concurrency

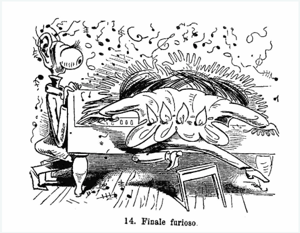
Sequential





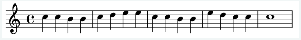
Concurrent





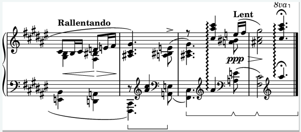
Concurrency

Sequential



6. Piano.

Concurrent



14. Finale furioso.

Doing things at the “same” time may lead to chaos

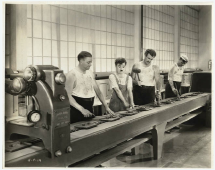
Necessary

communication & synchronization

Communication & synchronization

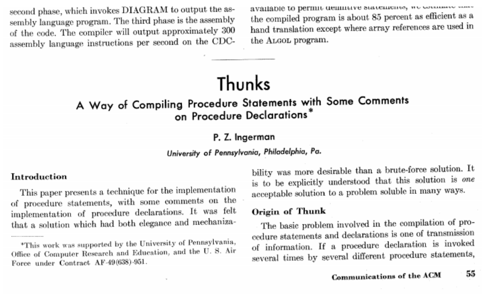
orch., coord.

pipelining



deep copying

Futures: past, present, and future



second phase, which invokes DIAGRAM to output the assembly language program. The third phase is the assembly of the code. The compiler will output approximately 300 assembly language instructions per second on the CDC-

available to permit definitive statements, we estimate that the compiled program is about 85 percent as efficient as a hand translation except where array references are used in the ALGOL program.

Thunks

A Way of Compiling Procedure Statements with Some Comments on Procedure Declarations*

P. Z. Ingerman

University of Pennsylvania, Philadelphia, Pa.

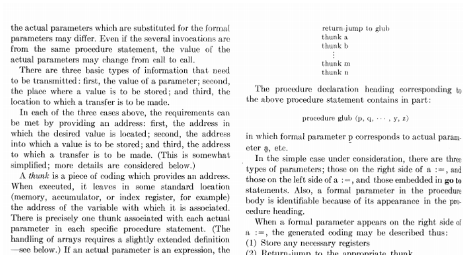
Introduction

This paper presents a technique for the implementation of procedure statements, with some comments on the implementation of procedure declarations. It was felt that a solution which had both elegance and mechanizability was more desirable than a brute-force solution. It is to be explicitly understood that this solution is *one* acceptable solution to a problem soluble in many ways.

*This work was supported by the University of Pennsylvania, Office of Computer Research and Education, and the U. S. Air Force under Contract AF-49(638)-951.

Origin of Thunk

The basic problem involved in the compilation of procedure statements and declarations is one of transmission of information. If a procedure declaration is invoked several times by several different procedure statements,



the actual parameters which are substituted for the formal parameters may differ. Even if the several invocations are from the same procedure statement, the value of the actual parameters may change from call to call.

There are three basic types of information that need to be transmitted: first, the value of a parameter; second, the place where a value is to be stored; and third, the location to which a transfer is to be made.

In each of the three cases above, the requirements can be met by providing an address: first, the address in which the desired value is located; second, the address into which a value is to be stored; and third, the address to which a transfer is to be made. (This is somewhat simplified; more details are considered below.)

A *thunk* is a piece of coding which provides an address. When executed, it leaves in some standard location (memory, accumulator, or index register, for example) the address of the variable with which it is associated. There is precisely one thunk associated with each actual parameter in each specific procedure statement. (The handling of arrays requires a slightly extended definition —see below.) If an actual parameter is an expression, the

```
return-jump to glub
thunk a
thunk b
   ⋮
thunk m
thunk n
```

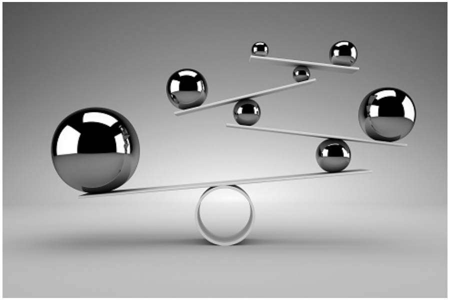
The procedure declaration heading corresponding to the above procedure statement contains in part:

```
procedure glub (p, q, ··· , y, z)
```

in which formal parameter p corresponds to actual parameter a, etc.

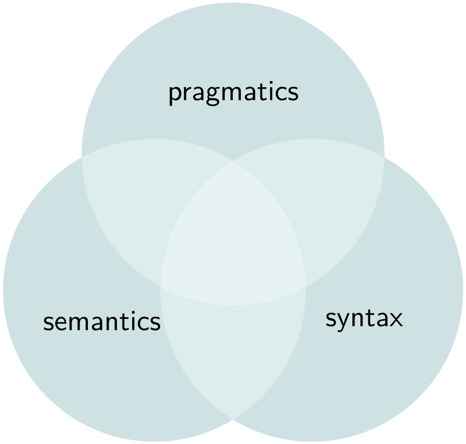
In the simple case under consideration, there are three types of parameters; those on the right side of a :=, and those on the left side of a :=, and those embedded in **go to** statements. Also, a formal parameter in the procedure body is identifiable because of its appearance in the procedure heading.

When a formal parameter appears on the right side of a :=, the generated coding may be described thus:
(1) Store any necessary registers
(2) Return-jump to the appropriate thunk



A balancing act

Aspects (among others)



$+ \ldots$

Further abstractions: “types”

- types & concurrency
- safety
- sharing & mutating
- many design decision there, as well
- capabilities

That’s it for now

UPPSALA UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1986

Abstractions To Control The Future

FRANCISCO RAMÓN FERNÁNDEZ REYES

Typing of otherwise (p1-113)

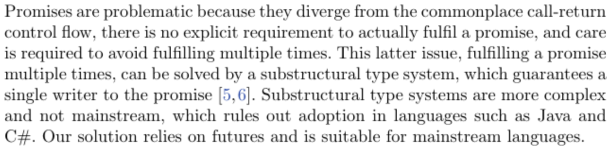
$$\text{(TS-OTHERWISE)}\quad \frac{\Gamma \vdash_{\rho} e_1 : Par\ \tau \qquad \Gamma \vdash_{\rho} e_2 : Par\ \tau}{\Gamma \vdash_{\rho} e_1 \times e_2 : Par\ \tau}$$

INE)

Multi-core (p1-117)

Meseguer et al. [1] used rewriting logic semantics and Maude to provide a distributed implementation of Orc. Their focus on the semantic model allows them to model check Orc programs. In this paper, our semantics is more fine-grained, and guides the implementation in a multicore setting.

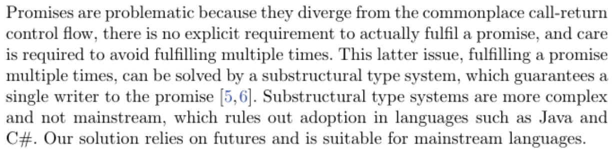
Substrural type systems not useful (p2:164)



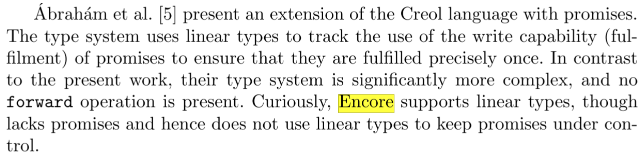
Promises are problematic because they diverge from the commonplace call-return control flow, there is no explicit requirement to actually fulfil a promise, and care is required to avoid fulfilling multiple times. This latter issue, fulfilling a promise multiple times, can be solved by a substructural type system, which guarantees a single writer to the promise [5,6]. Substructural type systems are more complex and not mainstream, which rules out adoption in languages such as Java and C#. Our solution relies on futures and is suitable for mainstream languages.

- bisims
- translational fa?, fa-compilation
- semantics

Substrural type systems not useful (p2:164)

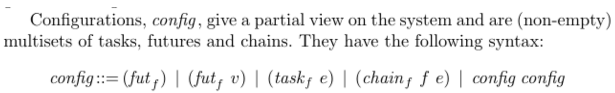

Promises are problematic because they diverge from the commonplace call-return control flow, there is no explicit requirement to actually fulfil a promise, and care is required to avoid fulfilling multiple times. This latter issue, fulfilling a promise multiple times, can be solved by a substructural type system, which guarantees a single writer to the promise [5,6]. Substructural type systems are more complex and not mainstream, which rules out adoption in languages such as Java and C#. Our solution relies on futures and is suitable for mainstream languages.

⋮


Ábrahám et al. [5] present an extension of the Creol language with promises. The type system uses linear types to track the use of the write capability (fulfilment) of promises to ensure that they are fulfilled precisely once. In contrast to the present work, their type system is significantly more complex, and no `forward` operation is present. Curiously, Encore supports linear types, though lacks promises and hence does not use linear types to keep promises under control.

- bisims
- translational fa?, fa-compilation
- semantics

Empty config (p2:165)



Configurations, *config*, give a partial view on the system and are (non-empty) multisets of tasks, futures and chains. They have the following syntax:

$$config ::= (fut_f) \mid (fut_f\ v) \mid (task_f\ e) \mid (chain_f\ f\ e) \mid config\ config$$

Evaluation order (p2:170)

$$\begin{aligned} E ::= {} & \bullet \mid E\,e \mid v\,E \mid E; e \mid \mathtt{get}\,E \mid \mathtt{fulfil}(E, e) \mid \mathtt{fulfil}(v, E) \\ & \mid \mathtt{Task}(E, e) \mid \mathtt{Chain}(e, E, e) \mid \mathtt{Chain}(E, v, e) \mid \mathtt{Chain}(v, v, E) \\ & \mid \mathtt{if}\ E\ \mathtt{then}\ e\ \mathtt{else}\ e \end{aligned}$$

Chaining syntax (p2:166,171)

Source language

(Red-Chain-Create)

$$\frac{\textit{fresh}\ g}{(\textit{task}_f\ E[h \overset{x}{\rightsquigarrow} e]) \rightarrow (\textit{fut}_g)\ (\textit{chain}_g\ h\ \lambda x.e)\ (\textit{task}_f\ E[g])}$$

Target language

(RI-Chain)

$$(\texttt{task}\ E[\texttt{Chain}(f, g, (\lambda x.e))]) \rightarrow (\texttt{chain}\ g\ e[f/x])\ (\texttt{task}\ E[f])$$

Double-write errors (p2-173)

$$\text{(TI-Constant)}\ \dfrac{\text{c is a constant of type } \tau}{\Gamma \vdash c : \tau} \qquad \text{(TI-Promise)}\ \dfrac{f\ : Prom\ \tau \in \Gamma}{\Gamma \vdash f : Prom\ \tau} \qquad \text{(TI-Variable)}\ \dfrac{x\ :\ \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{(TI-Unit)}\ \dfrac{}{\Gamma \vdash () : \mathtt{unit}}$$

$$\text{(TI-Stop)}\ \dfrac{}{\Gamma \vdash \mathtt{stop} : \tau} \qquad \text{(TI-Promise-New)}\ \dfrac{}{\Gamma \vdash \mathtt{Prom} : Prom\ \tau} \qquad \text{(TI-If)}\ \dfrac{\Gamma \vdash e : bool \quad \Gamma \vdash e' : \tau \quad \Gamma \vdash e'' : \tau}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ e'\ \mathtt{else}\ e'' : \tau}$$

$$\text{(TI-Statement)}\ \dfrac{\Gamma \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 ; e_2 : \tau} \qquad \text{(TI-Abstraction)}\ \dfrac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \qquad \text{(TI-App)}\ \dfrac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau}$$

$$\text{(TI-Fulfil)}\ \dfrac{\Gamma \vdash e : Prom\ \tau \quad \Gamma \vdash e' : \tau}{\Gamma \vdash \mathtt{fulfil}(e, e') : \mathtt{unit}} \qquad \text{(TI-Task)}\ \dfrac{\Gamma \vdash e : Prom\ \tau \quad \Gamma \vdash e' : Prom\ \tau \to\ \tau'}{\Gamma \vdash \mathtt{Task}(e, e') : Prom\ \tau}$$

$$\text{(TI-Get)}\ \dfrac{\Gamma \vdash e : Prom\ \tau}{\Gamma \vdash \mathtt{get}\ e : \tau} \qquad \text{(TI-Chain)}\ \dfrac{\Gamma \vdash e : Prom\ \tau \quad \Gamma \vdash e' : Prom\ \tau' \quad \Gamma \vdash e'' : Prom\ \tau \to \tau' \to\ \tau''}{\Gamma \vdash \mathtt{Chain}(e, e', e'') : Prom\ \tau}$$

$$\text{(Prom)}\ \dfrac{f \in dom(\Gamma)}{\Gamma \vdash (prm_f)\ ok} \qquad \text{(F-Prom)}\ \dfrac{f : Prom\ \tau \in \Gamma}{\Gamma \vdash (prm_f\ v)\ ok} \qquad \text{(Chain-Target)}\ \dfrac{\Gamma \vdash f : Prom\ \tau \qquad \Gamma \vdash e : \tau \to \tau''}{\Gamma \vdash (\mathtt{chain}\ f\ e)\ ok}$$

$$\text{(Task-Target)}\ \dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash (\mathtt{task}\ e)\ ok} \qquad \text{(Config-Target)}\ \dfrac{\Gamma \vdash config_1\ ok \qquad \Gamma \vdash config_2\ ok}{\Gamma \vdash config_1\ config_2\ ok}$$

Double-write errors (p2-173)

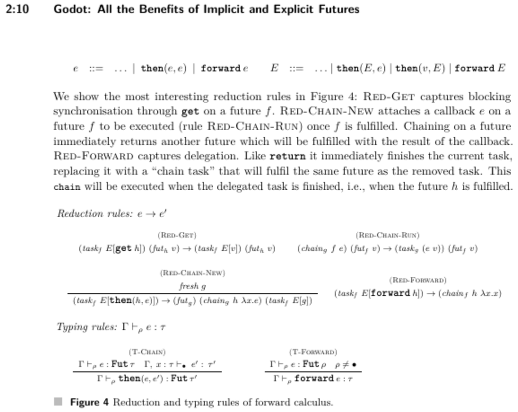
Source

$$\frac{\Gamma \vdash \mathit{config}_1\,\mathit{ok} \quad \Gamma \vdash \mathit{config}_2\,\mathit{ok} \qquad \mathit{defs}(\mathit{config}_1) \cap \mathit{defs}(\mathit{config}_2) = \varnothing \qquad \mathit{writers}(\mathit{config}_1) \cap \mathit{writers}(\mathit{config}_2) = \varnothing}{\Gamma \vdash \mathit{config}_1\ \mathit{config}_2\ \mathit{ok}}\ (\textsc{Config})$$

Target

$$\frac{\Gamma \vdash \mathit{config}_1\ \mathit{ok} \qquad \Gamma \vdash \mathit{config}_2\ \mathit{ok}}{\Gamma \vdash \mathit{config}_1\ \mathit{config}_2\ \mathit{ok}}\ (\textsc{Config-Target})$$

Syntax of chaining (p3 2:10)



2:10 Godot: All the Benefits of Implicit and Explicit Futures

$$e \;::=\; \dots \mid \texttt{then}(e,e) \mid \texttt{forward}\, e \qquad E \;::=\; \dots \mid \texttt{then}(E,e) \mid \texttt{then}(v,E) \mid \texttt{forward}\, E$$

We show the most interesting reduction rules in Figure 4: Red-Get captures blocking synchronisation through **get** on a future f. Red-Chain-New attaches a callback e on a future f to be executed (rule Red-Chain-Run) once f is fulfilled. Chaining on a future immediately returns another future which will be fulfilled with the result of the callback. Red-Forward captures delegation. Like **return** it immediately finishes the current task, replacing it with a "chain task" that will fulfil the same future as the removed task. This `chain` will be executed when the delegated task is finished, i.e., when the future h is fulfilled.

Reduction rules: $e \rightarrow e'$

(Red-Get)

$$(task_f\ E[\texttt{get}\, h])\ (fut_h\ v) \rightarrow (task_f\ E[v])\ (fut_h\ v)$$

(Red-Chain-Run)

$$(chain_g\ f\ e)\ (fut_f\ v) \rightarrow (task_g\ (e\ v))\ (fut_f\ v)$$

(Red-Chain-New)

$$\frac{fresh\ g}{(task_f\ E[\texttt{then}(h,e)]) \rightarrow (fut_g)\ (chain_g\ h\ \lambda x.e)\ (task_f\ E[g])}$$

(Red-Forward)

$$(task_f\ E[\texttt{forward}\, h]) \rightarrow (chain_f\ h\ \lambda x.x)$$

Typing rules: $\Gamma \vdash_\rho e : \tau$

(T-Chain)

$$\frac{\Gamma \vdash_\rho e : \texttt{Fut}\,\tau \quad \Gamma, x : \tau \vdash_\bullet e' : \tau'}{\Gamma \vdash_\rho \texttt{then}(e, e') : \texttt{Fut}\,\tau'}$$

(T-Forward)

$$\frac{\Gamma \vdash_\rho e : \texttt{Fut}\,\rho \quad \rho \neq \bullet}{\Gamma \vdash_\rho \texttt{forward}\, e : \tau}$$

Figure 4 Reduction and typing rules of forward calculus.

Abstractions to control the future

Martin Steffen

Typing for chaining

P2 (168)

(T-CHAIN)

$$\frac{\Gamma \vdash_{\rho} e : \mathit{Fut}\ \tau \quad \Gamma,\ x : \tau \vdash_{\tau'}\ e' : \tau'}{\Gamma \vdash_{\rho} e \overset{x}{\rightsquigarrow} e' : \mathit{Fut}\ \tau'}$$

P3 (2-10)

(T-CHAIN)

$$\frac{\Gamma \vdash_{\rho} e : \mathbf{Fut}\ \tau \quad \Gamma,\ x : \tau \vdash_{\bullet}\ e' : \tau'}{\Gamma \vdash_{\rho} \mathbf{then}(e, e') : \mathbf{Fut}\ \tau'}$$

Typing rule for the empty config? (p3 2:12)

$$\text{(T-EMPTY)}\quad \frac{}{\Gamma \vdash \diamond\ ok}$$

$$\text{(T-GCONFIG)}\quad \frac{\Gamma \vdash \mathit{config}\ \ ok \qquad \mathit{dom}(\Gamma) = \mathit{defs}(\mathit{config})}{\Gamma \vdash \mathit{config}}$$

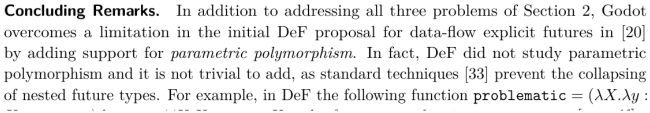
- normalization

Well-formedness

$$\frac{}{\vdash \epsilon}\ \text{(TF-ENV)} \qquad \frac{x \notin dom(\Gamma) \qquad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau}\ \text{(TF-ENVEXPR)} \qquad \frac{X \notin dom(\Gamma) \qquad \vdash \Gamma}{\vdash \Gamma, X}\ \text{(TF-ENVVAR)} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathcal{K}}\ \text{(TF-K)}$$

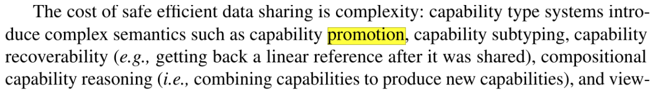
Prevent the collapse of the futures (p2 2:22)

Prevent the collapse of the futures (p2 2:22)


Concluding Remarks. In addition to addressing all three problems of Section 2, Godot overcomes a limitation in the initial DeF proposal for data-flow explicit futures in [20] by adding support for *parametric polymorphism*. In fact, DeF did not study parametric polymorphism and it is not trivial to add, as standard techniques [33] prevent the collapsing of nested future types. For example, in DeF the following function `problematic` $= (\lambda X.\lambda y :$

Promotion

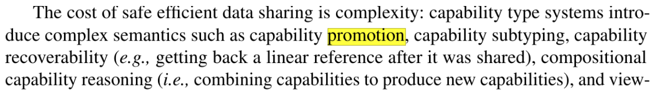
- what's (capability) promotion? $C^{\#}$

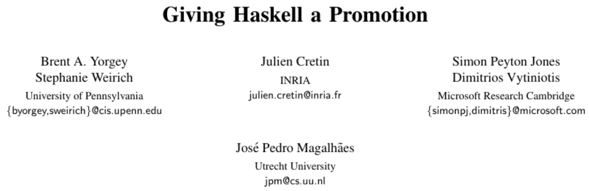


The cost of safe efficient data sharing is complexity: capability type systems introduce complex semantics such as capability promotion, capability subtyping, capability recoverability (*e.g.,* getting back a linear reference after it was shared), compositional capability reasoning (*i.e.,* combining capabilities to produce new capabilities), and view-

Promotion

- what's (capability) promotion? C$^{\#}$


The cost of safe efficient data sharing is complexity: capability type systems introduce complex semantics such as capability promotion, capability subtyping, capability recoverability (*e.g.,* getting back a linear reference after it was shared), compositional capability reasoning (*i.e.,* combining capabilities to produce new capabilities), and view-


Giving Haskell a Promotion

Brent A. Yorgey
Stephanie Weirich
University of Pennsylvania
{byorgey,sweirich}@cis.upenn.edu

Julien Cretin
INRIA
julien.cretin@inria.fr

Simon Peyton Jones
Dimitrios Vytiniotis
Microsoft Research Cambridge
{simonpj,dimitris}@microsoft.com

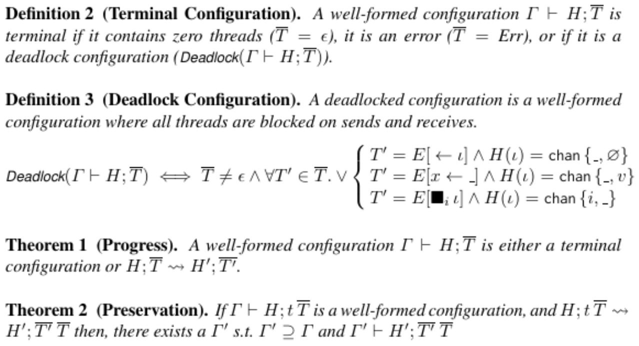
José Pedro Magalhães
Utrecht University
jpm@cs.uu.nl

A-normal form/SSA (p4:15)

(R-LET)

$$\frac{x \notin dom(H)}{H; \texttt{let}\, x \;=\; v\, \texttt{in}\, t \rightsquigarrow H, x \mapsto v; t}$$

Progress



Definition 2 (Terminal Configuration). *A well-formed configuration $\Gamma \vdash H; \overline{T}$ is terminal if it contains zero threads ($\overline{T} = \epsilon$), it is an error ($\overline{T} = Err$), or if it is a deadlock configuration ($\mathsf{Deadlock}(\Gamma \vdash H; \overline{T})$).*

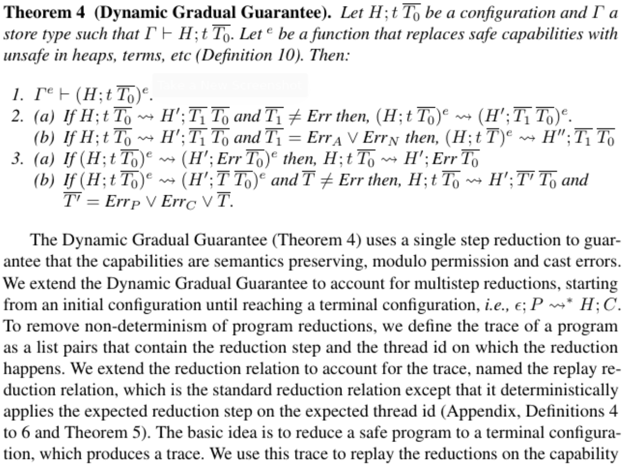
Definition 3 (Deadlock Configuration). *A deadlocked configuration is a well-formed configuration where all threads are blocked on sends and receives.*

$$\mathsf{Deadlock}(\Gamma \vdash H; \overline{T}) \iff \overline{T} \neq \epsilon \wedge \forall T' \in \overline{T}. \vee \begin{cases} T' = E[\leftarrow \iota] \wedge H(\iota) = \mathsf{chan}\,\{\_, \varnothing\} \\ T' = E[x \leftarrow \_] \wedge H(\iota) = \mathsf{chan}\,\{\_, v\} \\ T' = E[\blacksquare_i\, \iota] \wedge H(\iota) = \mathsf{chan}\,\{i, \_\} \end{cases}$$

Theorem 1 (Progress). *A well-formed configuration $\Gamma \vdash H; \overline{T}$ is either a terminal configuration or $H; \overline{T} \rightsquigarrow H'; \overline{T'}$.*

Theorem 2 (Preservation). *If $\Gamma \vdash H; t\,\overline{T}$ is a well-formed configuration, and $H; t\,\overline{T} \rightsquigarrow H'; \overline{T'}\,\overline{T}$ then, there exists a Γ' s.t. $\Gamma' \supseteq \Gamma$ and $\Gamma' \vdash H'; \overline{T'}\,\overline{T}$*

Dynamic gradual guarantee (p4:21)

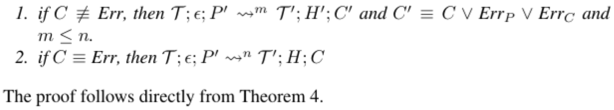


Theorem 4 (Dynamic Gradual Guarantee). *Let $H; t\,\overline{T_0}$ be a configuration and Γ a store type such that $\Gamma \vdash H; t\,\overline{T_0}$. Let ^{e} be a function that replaces safe capabilities with unsafe in heaps, terms, etc (Definition 10). Then:*

1. $\Gamma^e \vdash (H; t\,\overline{T_0})^e$.
2. (a) *If* $H; t\,\overline{T_0} \rightsquigarrow H'; \overline{T_1}\,\overline{T_0}$ *and* $\overline{T_1} \neq \mathit{Err}$ *then,* $(H; t\,\overline{T_0})^e \rightsquigarrow (H'; \overline{T_1}\,\overline{T_0})^e$.
   (b) *If* $H; t\,\overline{T_0} \rightsquigarrow H'; \overline{T_1}\,\overline{T_0}$ *and* $\overline{T_1} = \mathit{Err}_A \vee \mathit{Err}_N$ *then,* $(H; t\,\overline{T})^e \rightsquigarrow H''; \overline{T_1}\,\overline{T_0}$
3. (a) *If* $(H; t\,\overline{T_0})^e \rightsquigarrow (H'; \mathit{Err}\,\overline{T_0})^e$ *then,* $H; t\,\overline{T_0} \rightsquigarrow H'; \mathit{Err}\,\overline{T_0}$
   (b) *If* $(H; t\,\overline{T_0})^e \rightsquigarrow (H'; \overline{T}\,\overline{T_0})^e$ *and* $\overline{T} \neq \mathit{Err}$ *then,* $H; t\,\overline{T_0} \rightsquigarrow H'; \overline{T'}\,\overline{T_0}$ *and* $\overline{T'} = \mathit{Err}_P \vee \mathit{Err}_C \vee \overline{T}$.

The Dynamic Gradual Guarantee (Theorem 4) uses a single step reduction to guarantee that the capabilities are semantics preserving, modulo permission and cast errors. We extend the Dynamic Gradual Guarantee to account for multistep reductions, starting from an initial configuration until reaching a terminal configuration, *i.e.,* $\epsilon; P \rightsquigarrow^* H; C$. To remove non-determinism of program reductions, we define the trace of a program as a list pairs that contain the reduction step and the thread id on which the reduction happens. We extend the reduction relation to account for the trace, named the replay reduction relation, which is the standard reduction relation except that it deterministically applies the expected reduction step on the expected thread id (Appendix, Definitions 4 to 6 and Theorem 5). The basic idea is to reduce a safe program to a terminal configuration, which produces a trace. We use this trace to replay the reductions on the capability

Dynamic gradual guarantee (p4:21)



1. *if* $C \not\equiv Err$, *then* $\mathcal{T}; \epsilon; P' \rightsquigarrow^m \mathcal{T}'; H'; C'$ *and* $C' \equiv C \vee Err_P \vee Err_C$ *and* $m \leq n$.
2. *if* $C \equiv Err$, *then* $\mathcal{T}; \epsilon; P' \rightsquigarrow^n \mathcal{T}'; H; C$

The proof follows directly from Theorem 4.

Type inference

$$\text{(TI-S{\scriptsize TOP})} \quad \frac{}{\Gamma \vdash \texttt{stop} : \tau} \qquad \text{(TI-P{\scriptsize ROMISE}-N{\scriptsize EW})} \quad \frac{}{\Gamma \vdash \texttt{Prom} : \mathit{Prom}\ \tau} \qquad \frac{\Gamma \vdash e}{\Gamma}$$

$$\text{(TI-S{\scriptsize TATEMENT})} \quad \frac{\Gamma \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1; e_2 : \tau} \qquad \text{(TI-A{\scriptsize BSTRACTION})} \quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}$$

- Curry-style formulation

Type inference

$$\text{(TI-S{\scriptsize TOP})}\quad \frac{}{\Gamma \vdash \mathtt{stop} : \tau} \qquad \text{(TI-P{\scriptsize ROMISE}-N{\scriptsize EW})}\quad \frac{}{\Gamma \vdash \mathtt{Prom} : \mathit{Prom}\ \tau} \qquad \frac{\Gamma \vdash e}{\Gamma}$$

$$\text{(TI-S{\scriptsize TATEMENT})}\quad \frac{\Gamma \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1; e_2 : \tau} \qquad \text{(TI-A{\scriptsize BSTRACTION})}\quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}$$

- Curry-style formulation
- System F

Costs of abstractions

Buzz-word

Zero-cost abstractions

Further questions

- weak memory models
- compositional typing

References I

Bibliography

[1] Fernández Reyes, F. R. (2012). *Abstractions to Control the Future*. PhD thesis, Uppsala Universitet.

[2] Fernandez-Reyes, K., , Clarke, D., Castegren, E., and Vo, H.-P. (2018). Forward to a promising future. In *Proceedings of the 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018.*, volume 10852 of *Lecture Notes in Computer Science*, pages 162–180. Springer Verlag.

[3] Fernandez-Reyes, K., Clarke, D., Henrio, L., Johnsen, E. B., and Wrigstad, T. (2019). Godot: All the benefits of implicit and explicit futures. In Donaldson, A. F., editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPIcs*, pages 2:1–2:28. Schloss Dagstuhl — Leibniz-Zentrum für Informatik.

[4] Fernandez-Reyes, K., Clarke, D., and McCain, D. S. (2016). ParT: An asynchronous parallel abstraction for speculative pipeline computations. In Lluch-Lafuente, A. and Proença, J., editors, *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9686 of *Lecture Notes in Computer Science*, pages 101–120. Springer Verlag.

[5] Fernandez-Reyes, K., Noble, J., Gariano, I. O., Greenwood-Thessman, E., Michael, H., and Wrigstad, T. (2020). Dala: A simple capability-based dynamic language design for data-race freedom. check.